Table des matières

Chapitre 1

1 Rappel : les systèmes de numération

1.1 Le binaire

On a choisis d'associer au bit le plus à droite d'un paquet de bit, aussi appelé bit de poids faible, la valeur 2^0 = 1, le bit venant à sa gauche représentant une valeur deux fois plus élevée, 2^1 = 2 ainsi de suite....jusqu'au bit le plus à gauche, aussi appelé le bit de poids fort.

EXEMPLE:

Puissance de 2 associée	3	2	1	0
Valeur correspondante	2^3=8	2^2=4	2^1=2	2^0=1
Une valeur binaire Vaut alors	1 (1*8)	0 (0*4)	1 (1*2)	1 (1*1)
==>	8 + 2 + 1 = 11			

à la valeur binaire 1011 correspond donc la valeur décimale 11.

Par la suite pour éviter, tout malentendu on notera les valeurs binaires avec un % devant, 11 (décimal) s'écrira donc %1011 en binaire.
Cette notation est utilisée dans de nombreux langages de programmation (pour l'assembleur, on fera suivre la valeur binaire de b , 11 (décimal) s'écrira 1011b).

Si on regroupe 8 variables logiques (8 bits) on pourra représenter des nombres qui évolueront entre:
- %00000000 = 0 en décimal
- %11111111 = 128+64+32+16+8+4+2+1 = 255 en décimal.

Nous verrons par la suite les problèmes qui apparaissent pour coder les nombres négatifs.

1.2 L'hexadécimal

Tous comme on peut dire que le binaire constitue un système de codage en base 2 (0 ou 1 cela fait deux états possibles), le décimal un système en base 10 (0 1 2 3 4 5 6 7 8 9 => 10 états possibles) vous l'aurez compris l'hexadécimal constitue un système base 16, pour représenter les 10 première valeurs, la notations est identique à celle du système décimal et pour les 6 autres on prend les 6 premières lettres de l'alphabet.

Pour des soucis de clarté on fait suivre toute notation hexadécimale de la lettre h (convention de l'assembleur, pour le langage C il faudra faire précéder la notation de 0x , dans le basic on fait précéder la notation de $ ou de #$ dans certains basic 8 bits), ce qui donne :

```
Décimal        Binaire        Hexadécimal

00             %0000          0h
01             %0001          1h
02             %0010          2h
03             %0011          3h
04             %0100          4h
05             %0101          5h
06             %0110          6h
07             %0111          7h
08             %1000          8h
09             %1001          9h
10             %1010          Ah
11             %1011          Bh
12             %1100          Ch
13             %1101          Dh
14             %1110          Eh
15             %1111          Fh
```

Vous le voyez l'hexadécimal comporte de nombreux avantages, on peut représenter 16 valeurs avec seulement 1 chiffre, alors qu'il en faut 2 pour la décimal et 4 pour le binaire.

Le fait que l'Hexadécimal soit une base 16 = 2^4 permet de convertir n'importe quel quarté de bit en un chiffre hexadécimal. On peut tout comme avec le binaire, écrire des nombres hexadécimaux en groupant des chiffres ainsi.

Puissance de 16 associée	3	2	1	0
Valeur correspondante	16^3=4096	16^2=256	16^1=16	16^0=1
Valeur Hexadécimale Vaut alors	2 (2*4096)	A (10*256)	B (11*16)	3 (3*1)
==>	8192 + 2560 + 176 + 3 = 10931			

1.3 Opérations Arithmétiques

On peut tout comme avec le décimal effectuer des opérations standards tout comme l'addition, la soustraction, ou la multiplication:

1.3.1 L'addition :

en décimal (un rappel sûrement inutile mais qui clarifie des choses...)

```
Retenue              11
1er nombre          127
2eme nombre       +  96
résultat            223
```

En binaire on procède de la même façon en respectant les règles suivantes:

```
%0+%0=%0
%0+%1=%1
%1+%0=%1
1%+%1=%0 et je retient 1
```

```
Retenue            11
ce qui donne pour  127   =>    %01111111
                 +  96   =>  + %01100000
Résultat         = 223   =>  = %11011111
```

De même en hexadécimal

```
  127   =>     7Fh
+  96   =>     60h
= 223   =>     DFh
```

Désolé cet exemple ne comporte pas de retenues.

En voici un qui en comporte:

```
        Retenue      1
                    30  => 1Eh
                 +  52  => 34h
                 =  82  => 52h
```

Pour ce qui est de la soustraction, étant donné que cela correspond à additionner l'opposé d'un nombre, nous verrons cela avec la codification des nombres négatifs.

1.3.2 Les dépassements dans l'addition:

prenons un exemple d'addition:

```
                111
   160          %10100000
 + 100          %01100100
 =   4          %00000100
```

Visiblement, le résultat est faux, du moins c'est ce que vous dirai votre petit frère de 6 ans qui excelle au cour préparatoire.
En regardant de plus prés, on peut se rendre compte qu'il nous reste une retenue, il faut donc en faire quelque chose...
L'idée immédiate que ce cas susciterai est la bonne, il suffit de créer un 9e bit, qui sera alors égal à la retenue, notre résultat s'écrit alors %100000100, ce qui donne 2^8=256 + 2^2=4 = 260 ce qui est le résultat attendu.

Comme nous le verrons, avec les microprocesseurs, il existe des instructions directement dédiées à la gestion de la retenue.

CONCLUSION: La somme de deux valeurs binaires codés sur un certain nombre de bit, peut dans certains cas créer un dépassement qui engendre une retenue.

Pour connaître le nombre de bits maximal que peut comporter le résultat, il suffit de prendre le nombre de bits de la valeur qui en compte le plus et de lui ajouter 1.

Exemple:

La somme d'un nombre codé sur 4 bits et d'un nombre codé sur 7 bits prendra donc au maximum 7+1=8 bits

Dans vos futurs programmes, il faudra toujours avoir cette notion en tête.

1.3.3 Les dépassements dans la multiplication:

La multiplication binaire, du moins pour les nombres positifs, se résout exactement comme la multiplication décimale, nous allons donc passer directement aux problèmes de dépassement.

Relation apprise au collège (a^x)*(a^y)=a^(x+y).

Nous savons aussi que l'on peut représenter sur p bits 2^p valeurs différentes, par exemple sur 8 bits nous pouvons représenter 256 valeurs différentes soit de 0 à 255.

Donc pour p bits la valeur maximale vaudra 2^p - 1.

si maintenant nous multiplions une valeur n1 sur p bits, par une valeur n2 sur q bits, on aura:

n1max = 2^p - 1
n2max = 2^q - 1

n1max * n2max = (2^p - 1)*(2^q - 1)

si on développe

=> 2^p*2^q-2^p-2^q = (2^(p+q)-2^p-2^q))
= (2^(p+q)-(2^p+2^q)).

On peut considérer (2^p+2^q étant petit devant 2^(p+q) mais grand par rapport à 1) que pour coder le résultat du produit d'un nombre représenté sur p bits par un nombre représenté sur q bit, il faudra p+q bits:

exemple numérique:

pour coder le produit de deux octets, il faudra 8+8=16 bits, soit un mot mémoire.

ex: 253 * 220 = 55660
=> FDh * DCh = D96Ch

1.4 Les nombres négatifs

Un nombre tel qu'on à appris à l'écrire précédemment est appelé un nombre <u>non signé</u> (unsigned en anglais)(il est toujours positif), maintenant nous allons apprendre à écrire des nombres qui peuvent représenter des valeurs négatives: On dira qu'ils sont <u>signés</u> (signed en anglais).

Les nombres négatifs obéissent à une règle de codage un peu spéciale appelée complémentation à 2.

Pour écrire un nombre négatif, 3 étapes:

a) on prend sa valeur absolue en binaire
ex:%00000100 = 4

b) on inverse l'état logique de tous les bits
0 devient 1 et 1 devient 0
ex:%00000100 =>%11111011

c) on lui ajoute 1
ex:%11111011+1=%11111100

%11111100 représente donc la valeur décimale -4.

<u>constatation :</u> le plus petit nombre que l'on peut coder sur 8 bits, est maintenant %10000000, soit -128 et le plus grand est %01111111 = +127
.
Si on à un nombre binaire comment savoir combien il vaut en décimal ?

Il suffit de faire la marche inverse.

Il faut savoir que le bit de poids le plus fort, est, dans un nombre signé le bit de signe, si il est à 0, il s'agit d'un nombre positif, sinon c'est un nombre négatif.

Donc 2 cas:

a)- Le bit de poids fort est à 0, le nombre est positif, on fait une conversion standards.

b)- Le bit de poids fort est à 1, le nombre est négatif alors:
- on lui soustrait 1
- on inverse l'état logique des bits
- on fait une conversion standards pour connaître la valeur absolue.

1.5 Les opérations arithmétiques avec les nombres négatifs

1.5.1 L'addition

L'addition reste inchangée, au détail prés que si il nous reste une retenue une fois le calcul terminé, elle à maintenant un rôle un peut spécial, nous devrons l'ignorer pour l'instant, nous reverrons cela avec les instructions du microprocesseur.

exemple:

```
retenue    11111100
     -4 => %11111100
    + 5 => %00000101
    = 1 => %00000001
```

Le résultat est visiblement correct, la procédure à suivre étant identique à celle des nombres non signés, les microprocesseurs n'ont besoin que d'une instruction pour traiter l'addition.

1.6 Les opérations logiques

Dans tous les microprocesseurs, il est possible d'effectuer les opérations logiques standards: ET, OU, OU exclusif, NON.

1.6.1 Le "ET" logique

Pour que le résultat soit vrai, il faudra avoir la première opérande vraie ET la seconde vraie. On peut considérer qu'il s'agit d'une multiplication logique.

Table de vérité:

0 ET 0 => 0
0 ET 1 => 0
1 ET 0 => 0
1 ET 1 => 1

1.6.2 Le "OU" logique (aussi appelé ou inclusif)

Pour que le résultat soit vrai, il faut, que la première opérande soit vraie, OU que la seconde opérande soit vraie.

Table de vérité:

0 OU 0 => 0
0 OU 1 => 1
1 OU 0 => 1
1 OU 1 => 1

1.6.3 Le "OU EXCLUSIF"

Pour que le résultat soit vrai, il faut, que la première opérande soit vraie, OU que la seconde opérande soit vraie, mais pas les deux.

Table de vérité:

0 OU EXCLUSIF 0 => 0
0 OU EXCLUSIF 1 => 1
1 OU EXCLUSIF 0 => 1
1 OU EXCLUSIF 1 => 0

1.6.4 Le "NON" aussi appelé complémentation

Cette opération logique se contente de renvoyer l'inverse logique de ce que l'on lui envoie.

NON 0 => 1
NON 1 => 0

1.7 Les décalages binaires

Les décalages binaires sont des opérations logiques un peu spéciales, que l'on peut généralement effectuer sur des octets, des mots, des doubles mots mais qui ont aucune signification sur un bit séparé.
Un décalage binaire consiste à faire "glisser" les bits d'un octet (ou mot ou double mot) vers la droite ou vers la gauche, nous verrons qu'il existe des décalages circulaires, aussi appelés rotations binaires, et des décalages arithmétiques.

conventions d'écriture:
(cette notation des celle adoptée dans le langage C).

B=<<A Signifiera: B est égal à A que l'on à décalé vers la gauche.
B=>>A Signifiera: B est égal à A que l'on à décalé vers la droite.

1.7.1 Le décalage binaire gauche

Faire un décalage binaire à gauche consiste à prendre chaque bit et lui faire occuper la place de celui qui se trouve à sa gauche. Le bit qui se trouve complètement à droite (ou bit de poids faible) se retrouve avec 0, le bit qui se trouve complètement à gauche ou bit de poids fort est éjecté de l'octet, il disparaît donc.

Exemple:

```
A =        %0  0  1  0  1  0  0  1
            /  /  /  /  /  /  /  /
      Ejecté /  /  /  /  /  /  /
B =<< A    %0  1  0  1  0  0  1  0
            /  /  /  /  /  /  /  /
      Ejecté /  /  /  /  /  /  /
C =<< B    %1  0  1  0  0  1  0  0
```

Dans un premier temps nous supposerons que ces valeurs binaires sont des valeurs non signées.

Si maintenant, nous évaluons la valeur décimale de A

A = 0*128+0*64+1*32+0*16+1*8+0*4+0*2+1*1= 32+8+1= 41

B = 0*128+1*64+0*32+1*16+0*8+0*4+1*2+0 = 64+16+2= 82

C = 1*128+0*64+1*32+0*16+0*8+4*1+0*2+0*1 = 128+32+4= 164

Nous remarquons que C=B*2 et que B=A*2 donc C=A*4, apparemment décaler une valeur binaire d'un bit vers la gauche, revient à la multiplier par 2.

Démonstration: Dans le cas général

A = a7*2^7 + a6*2^6 + a5*2^5 + a4*2^4 + a3*2^3 + a2*2^2 + a1*2^1 + a0*2^0

si maintenant nous multiplions par 2

2*A =2*(a7*2^7 + a6*2^6 + a5*2^5 + a4*2^4 + a3*2^3 + a2*2^2 + a1*2^1 + a0*2^0)

= 2*a7*2^7 + 2*a6*2^6 + 2*a5*2^5 + 2*a4*2^4 + 2*a3*2^3 + 2*a2*2^2 + 2*a1*2^1 + 2*a0*2^0

comme 2*2^n = 2^(n+1)

= a7*2^8 + a6*2^7 + a5*2^6 + a4*2^5 + a3*2^4 + a2*2^3 + a1*2^2 + a0*2^1

si maintenant nous comparons le résultat à A en identifiant termes à termes.

Même si vous n'avez pas bien suivis la démonstration, il faut retenir que:

Effectuer un décalage binaire à gauche, revient à multiplier ce nombre par 2.

Ce qui entraîne automatiquement

Effectuer n décalages binaires à gauche, revient à multiplier le nombre par 2^n.

Note: Vous pouvez faire l'essai si vous ne me croyez pas, mais:

Cela marche pour les nombres signés tous comme pour les nombres non signés. Il existe la dessus une certaine confusion dans le monde informatique, entraînée, je pense, par le fait qu'en assembleur il existe 2 instructions différentes pour décaler à gauche un nombre signé ou non signé, alors que c'est exactement la même chose.

1.7.2 Le décalage binaire droit

La marche à suivre pour réaliser un décalage binaire droit est similaire à celle utilisé pour décaler à gauche, à ceci prés que chaque bit ne prend plus la place du bit placé à sa gauche, mais celle du bit placé à sa droite. On injecte à gauche un bit toujours égal à 0, et le bit de poids faible est éjecté.

Exemple:

```
A =        %0  1  0  0  1  0  1  0
              \  \  \  \  \  \  \  \
               \  \  \  \  \  \  \  \
B =>> A    %0  0  1  0  0  1  0  1   Ejecté
              \  \  \  \  \  \  \  \
               \  \  \  \  \  \  \  \
C =>> B    %0  0  0  1  0  0  1  0   Ejecté
```

Vous l'aurez deviné, **effectuer un décalage binaire à droite, revient à diviser le nombre par 2.**

Par conséquent:
Effectuer n décalages binaires à droite, revient à diviser le nombre par 2^n.

Attention! cette opération logique n'est pas valable pour les nombres signés, vous comprendrez facilement que si le bit de signe passe de 1 à 0, le nombre devient un nombre positif, ce qui n'a plus rien à voir avec une division par 2. Nous verrons avec l'assembleur, que les microprocesseurs ont une instruction dédiée au cas des nombres négatifs, on parlera alors de décalage arithmétique droit.

1.7.3 A quoi ça sert ?

Il faut savoir que réaliser une multiplication, prend sur un microprocesseur plusieurs cycles d'horloge, voici les chiffres moyens pour les différents modèles de chez intel.

- 80386 → 26 cycles d'horloge.
- 80486 → 26 cycles d'horloge.
- pentium → 11 cycles d'horloge.

Pour une division c'est pire:

- 80386 → 38 cycles d'horloge.
- 80486 → 40 cycles d'horloge.
- pentium → 40 cycles d'horloge.

Ce qui signifie que si votre microprocesseur, par exemple un pentium 100, est cadencé à 100 Mhz, il pourra au maximum exécuter 100/11 soit environ 9 millions de multiplications par seconde, et environ 2.2 millions de divisions par secondes.

Voyons maintenant le nombre de cycles d'horloge que prend un décalage binaire à gauche.

- 80386 → 2 cycles d'horloge.
- 80486 → 3 cycles d'horloge.
- pentium → 1 cycle d'horloge.

Pour le décalage droit c'est la même chose.

Dans ce cas, vous pourrez réaliser sur votre pentium 100, 100/1 = 100 millions de multiplications par secondes. Ou 100 millions de divisions par secondes.

Malheureusement, cela ne marche que pour les multiplications sur des puissances de 2, mais dans certains cas on peut s'adapter par exemple:
(cet exemple est réel, il est fortement utilisé dans les animations en vga)

pour multiplier par 320:

320 n'est pas une puissance de 2, mais il peut s'écrire:
320 = 256+64 = 2^8+2^6.

Si on veut que B contienne 320*A soit B=A*320

on a : B = A*(256+64) = A*256 + A*64 = A*2^8+A*2^6

Donc il suffit de décaler A à gauche 8 fois, de garder le résultat et de décaler A à droite 2 fois, et d'additionner le résultat au précèdent. On aura multiplier A par 320 en:

1 cycle pour décaler A de 8 positions à gauche
+1 cycle pour mémoriser le résultat dans B
+1 cycle pour décaler A de 2 positions à droite (8-2=6)
+1 cycle pour additionner le résultat a B
=4 cycles sur un pentium, au lieu de 11, on est toujours gagnant.

1.7.4 La rotation binaire gauche

La rotation binaire gauche est similaire au décalage binaire gauche, au détail prés que les bits qui sortent par la gauche, sont injectés par la droite:

si A est un nombre binaire sur 8 bits

A = a7 a6 a5 a4 a3 a2 a1 a0

si on applique une rotation binaire gauche à A on aura

A = a6 a5 a4 a3 a2 a1 a0 a7

si on recommence, on aura

A = a5 a4 a3 a2 a1 a0 a7 a6

Vous l'aurez compris, si on effectue 8 rotations binaires gauches sur un octet, on sera revenu au point de départ . Au point de vue arithmétique la rotation binaire n'a aucune signification.

1.7.5 La rotation binaire droite

La rotation binaire droite, se réalise comme la rotation binaire gauche, mais dans l'autre sens: Le bit qui sort à droite est injecté à la gauche.

Chapitre 2

2 Introduction : les microprocesseurs INTEL

2.1 La mémoire

Une analogie consiste à comparer la mémoire à une longue rangée de tiroirs alignes les un derrière les autres. Si on donne à chaque tiroir un numéro, en commençant par 0 pour le 1er tiroir, on dira que ce numéro est l'adresse de la mémoire, dans la suite on parlera d'adresse mémoire . La coutume est (de nombreux avantages la justifie) de noter les adresses mémoires en hexadécimal.

Dans un P.C., l'unité de mémoire est l'octet, ce qui signifie que chaque tiroir comportera un octet, soit 8 bits. On peut donc imaginer que notre tiroir contient 8 paragraphes, chacun représentants un bit pouvant représenter deux états 0 ou 1.

Avoir un ordinateur qui possède 8Mo de mémoire signifie qu'il possède 1024*1024*8 octets de mémoire, et donc 1024*1024*8*8 bits de mémoire.

Dans un système informatique, les deux opérations possibles sur la mémoire sont l'écriture et la lecture d'une valeur. Dans n'importe quel langage informatique:

Pour écrire dans la mémoire on doit fournir deux paramètres au microprocesseur:

a) L'adresse mémoire ou l'on va écrire.
b) La valeur que l'on vas écrire à cette adresse (sur 8,16,ou 32bits).

Pour lire une valeur dans la mémoire on doit fournir ces deux paramètres

a) L'adresse mémoire ou on veut lire.
b) l'endroit ou on doit conserver la valeur lue (valeur sur 8, 16, ou 32 bits).

Nous verrons à la fin de ce cour , que pour lire ou écrire dans la mémoire on utilise l'instruction assembleur MOV.

2.2 Des segments pour se positionner dans la mémoire

Revenons à nos tiroirs; Imaginez maintenant que l'on mette une étiquette tous les 16 tiroirs, on inscrit sur la première étiquette la valeur 0, puis 16 tiroirs plus loin la valeur 1, ainsi de suite. on appellera le numéro de l'étiquette le segment, et la distance par rapport à cette étiquette l'offset. une adresse mémoire se composera donc de deux parties:

*Un segment toujours représenté en hexadécimal non signé
*Un offset toujours représenté en hexadécimal non signé

on notera l'adresse segment:offset.

exemple A000h:0000h

On peut trouver une correspondance (en sachant qu'il y a un segment tout les 16 octets) entre l'adresse sous la forme segment:offset, et l'adresse physique (on l'appelle ainsi car c'est cette adresse qui sera déposée sur le bus d'adresse du microprocesseur l'or d'un accès à la RAM).

adresse physique = SEGMENT*16+OFFSET.

Limitations: Une limitation importante de ce modèle est due à l'héritage 16 bit des microprocesseurs intel: Les segments et les offsets sont codés sur 16 bits non signés . De plus le bus d'adresse des premiers microprocesseurs étant sur 20 bits, l'adresse maximale possible était FFFFFh soit au maximum 1Mo adressable, cette limitation est restée et l'adresse maximale autorisée en mode réel est F000h:FFFFh

Ces considérations expliquent l'impossibilité qu'a le DOS d'utiliser la mémoire étendue, on se retrouve donc (quand on à enlevé la zone d'adressage du BIOS en ROM et la RAM vidéo) avec une mémoire vive adressable de 640Ko au maximum (mémoire conventionnelle).

L'utilisation des adresses sous la forme segment:offset, n'est pas un choix, c'est une contrainte que nous devrons supporter tant que nous programmerons en mode réel.

REMARQUE: Une remarque importante, il s'agit d'éviter une erreur présente dans la quasi totalité des livres sur la programmation assembleur, qui consiste à croire que la mémoire est divisée en segments de 64Ko, ce qui est faux.

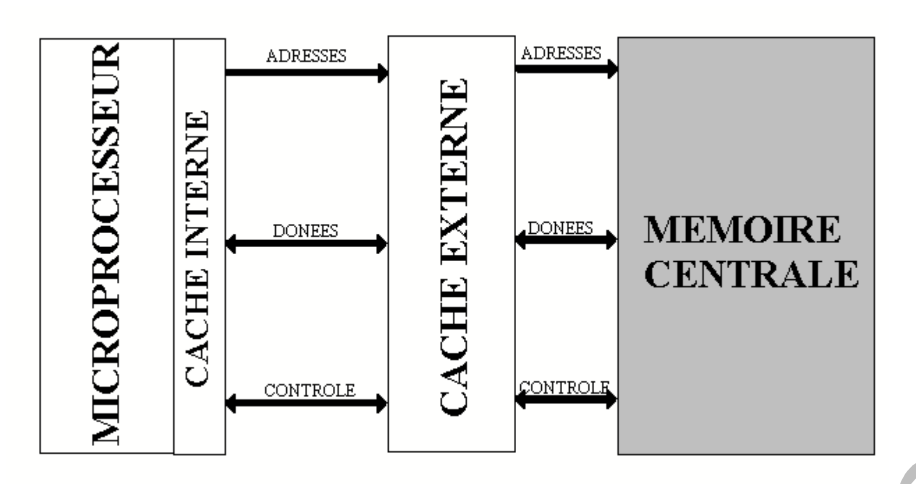
La mémoire est divisée en segments de 16 octets, à partir desquels il est possible de couvrir une zone de 64Ko à l'aide d'un offset de 16 bits.

AUTRE REMARQUE: A partir de segments distincts on peut accéder à une même adresse mémoire en effet ces adresses sont identiques:

```
segment offset
0005h:0001h ==>    5h*16+1h   = 80 + 1  = 81
0004h:0011h ==>    4h*16+11h  = 64 + 17 = 81
0003h:0021h ==>    3h*16+21h  = 48 + 33 = 81
0002h:0031h ==>    2h*16+31h  = 32 + 49 = 81
0001h:0041h ==>    1h*16+41h  = 16 + 65 = 81
0000h:0051h ==>    0h*16+51h  = 0  + 81 = 81
```

On dit alors que ces segments sont entrelacés.

2.3 Un microprocesseur pour agir sur la mémoire



2.3.1 LES BUS DE COMMUNICATION

Un bus est un groupement de conducteurs électriques, représentant chacun une variable logique, qui sert à faire transiter des informations entre plusieurs éléments.

2.3.1.1 Le bus d'adresse

Ce bus, d'une taille de 20 bit sur un PC XT, à été porté à une taille de 32 bits à partir du 386. Sa tache est de fournir l'adresse visée par le microprocesseur au contrôleur de mémoire. Ce bus est unidirectionnel, l'information ne vas que dans le sens microprocesseur --> périphérique.

2.3.1.2 Le bus de données

C'est par ce bus que le microprocesseur transmet des données à ses circuits périphériques, ce bus de 8 bits dans le 8088 à été porté à 32 bits dans le 386 puis à 64 bits dans le pentium. Ce bus est bidirectionnel, il permet au microprocesseur de lire des données, tous comme, il lui permet d'en écrire.

2.3.1.3 Le bus de contrôle

C'est par ce bus que vas être communiqué le sens du transfert sur le bus de données (lecture ou écriture) ainsi que la taille du transfert (8, 16, 32, 64 bits). C'est aussi par ce bus que le circuit périphérique dira au microprocesseur s'il est prêt à émettre ou recevoir des données.

2.3.1.4 Mécanisme d'accès mémoire

Dans un premier temps nous ignorerons le rôle joué par la mémoire cache.

a)Accès en lecture: Le microprocesseur transmet par le bus d'adresse l'adresse de la mémoire dont il veut lire le contenu, il place sur le bus de contrôle la taille de la donnée qu'il veut lire, et l'ordre LECTURE. A partir de ce moment la, le microprocesseur scrute le bus de contrôle pour savoir si la donné est arrivée, ou si il faut encore attendre. Dés que le bus de contrôle informe le microprocesseur que la donnée est disponible, le microprocesseur lit la donnée sur son bus de donnée.

b)Accès en écriture: Le microprocesseur transmet par le bus d'adresse l'adresse de la mémoire ou il veut écrire, il place sur le bus de contrôle la taille de la donnée qu'il veut écrire ainsi que l'ordre ECRITURE. Il place sur le bus de données la donnée à écrire. A partir de ce moment la, le microprocesseur scrute le bus de contrôle pour savoir si la donnée à été écrite en mémoire. Dés que le bus de contrôle informe le microprocesseur que la donnée à été écrite, le microprocesseur peut continuer son travail.

2.3.1.5 Temps d'accès mémoire

C'est un chiffre exprimé généralement en ns (nano secondes) qui détermine le temps qui vas s'écouler entre la demande de lecture (ou d'écriture) mémoire et sa réalisation. On peut faire l'analogie entre un temps accès mémoire et le temps qu'il faut pour ouvrir un tiroir et ainsi accéder à son contenu. Avec de la mémoire à 80ns (la vielle RAM avant que l' EDO n'apparaisse), il s'écoule donc 80ns entre la demande

d'écriture (ou de lecture) et sa réalisation, cela peut vous paraître très peu, mais pourtant c'est énorme à l'échelle du microprocesseur.

En sachant qu'un pentium à 100Mhz, exécute une instruction (parfois 2) en 10ns, on à le temps d'exécuter 8 instructions en attendant une écriture mémoire, sur un pentium 200 c'est pire, on peut en exécuter environ 16. Dans les microprocesseurs anciens (avant le 486), le microprocesseur ne pouvait rien faire en attendant l'écriture, il bloquait, maintenant on peut exécuter des instructions en parallèle avec une demande d'accès mémoire.

2.3.2 LA MEMOIRE CACHE

2.3.2.1 La mémoire cache interne

Il s'agit d'une mémoire un peut spéciale, quand à sa constitution, à son rôle, et à l'endroit ou elle se trouve.

Reprenons nos fameux tiroirs:
Vous êtes donc dans votre bureau, au fond de la pièce, une grande armoire qui comporte tous les tiroirs dont nous parlions précédemment. Votre entreprise ayant réalisé de gros bénéfices vous avez acheté un grand bureau et employé une secrétaire.

La taille de votre bureau vous permet de poser dessus environs une vingtaine de dossiers en plus de votre espace de travail. Si vous avez besoin d'un dossier particulier, dans un premier temps vous regardez s'il n'est pas sur votre bureau, s'il y est vous le prenez. Si votre dossier n'est pas sur le bureau, vous allez demander à votre secrétaire de vous l'apporter (car vous êtes très fainéant), Cette opération prendra un certain temps, et vous serez tenté de conserver ce dossier sur votre bureau au cas ou il serait à nouveau utile. La fin de la journée arrive et votre bureau est encombré de dossiers, il vous en faut un autre, vous appelez votre secrétaire qui vous l'apporte, et comme il ne reste plus de place sur votre bureau, vous faites ranger à votre secrétaire un dossier dont vous pensez ne plus avoir besoin.

Et bien la mémoire cache fonctionne exactement sur ce principe.
Vous êtes le microprocesseur, votre bureau la mémoire cache, et votre secrétaire le contrôleur mémoire qui gère vos tiroirs de mémoire.

Quand le microprocesseur à besoin d'une donnée stockée en mémoire:

Il regarde si cette donnée est dans le cache, si elle y est il la lit, si elle n'y est pas, il demande au contrôleur mémoire d'aller la chercher dans la mémoire centrale, il l'utilise alors tout en la gardant soigneusement dans le cache. Si maintenant le cache est plein, alors il supprime du cache la donnée qui y est restée le plus longtemps sans être utilisée, et donne la place disponible à la nouvelle donnée.

La mémoire cache interne, qui est comme son nom l'indique interne au microprocesseur, est très rapide d'accès, on peut considérer son temps de réponse comme nul. On peut se poser la question de savoir pourquoi toute la mémoire de l'ordinateur n'est pas de la mémoire cache, en fait la raison est d'ordre économique: La mémoire cache est encore très chère, un ordinateur composé uniquement de mémoire cache serait hors de prix. Un exemple intéressant de mémoire cache est celle du pentium pro dont la capacité à été portée à 256Ko, ce qui est énorme pour un cache interne.

Quelques chiffres:

MICROPROCESSEUR	BUS DE DONNEES	BUS D'ADRESSES	TAILLE DU CACHE
8088	8	20	0
8086	16	20	0
80286	16	24	0
80386SX	16	32	0
80386DX	32	32	0
80486SX	32	32	4Ko
80486DX	32	32	4Ko
80486DX2	32	32	8Ko
80486DX4	32	32	16Ko
PENTIUM	64	32	16Ko
CYRIX 686	64	32	16Ko
PENTIUM PRO	64	32	256Ko

2.3.2.2 La mémoire cache externe

Le volume de votre entreprise augmente, et décidément votre secrétaire devient insupportable, elle n'accepte plus de faire des centaines de trajets tiroirs - bureau par jour.

L'idée vous vient d'installer entre votre bureau et les tiroirs de l'armoire une table basse sur laquelle vous pourrez entreposer plus d'une centaine de dossiers. Ainsi avec cet ingénieux système les trajets bureau - tiroirs seront moins fréquents.

Si vous avez besoin d'un dossier, dans un premier temps vous regarderez s'il se trouve sur votre bureau, si c'est le cas vous le prenez et c'est fini.

Si votre dossier ne se trouve pas sur le bureau, vous pouvez regarder s'il n'est pas sur la table basse, s'il y est, il suffit de tendre le bras pour le prendre, ce qui est tout de même plus rapide que d'appeler la secrétaire. Et comble de malchance il n'est pas sur la table basse (il s'agit sûrement d'un nouveau dossier que vous n'avez pas regardé de la semaine), la secrétaire vous l'amènera avec le sourire.

Votre table basse remplis le même rôle que votre bureau, au détail près qu'elle est un peut plus difficile d'accès et qu'on peut entreposer dessus plus de dossiers.

Et bien dans un ordinateur le cache de second niveau rempli exactement le même rôle que la table basse de votre bureau, il permet de garder de coté des données auxquelles on accède relativement souvent. On pourrait très bien imaginer que dans le futur des ordinateurs contiennent des caches de troisième ou de quatrième niveau.

La taille des caches de second niveau à augmenté ces dernieres années, ils faisaient 32Ko sur les 386, 128Ko sur les 486, il font maintenant de 256Ko à 512Ko sur les pentiums les plus récents. La technologie mémoire qu'ils utilisent, permet, actuellement, d'accéder à une donnée en environ 20ns.

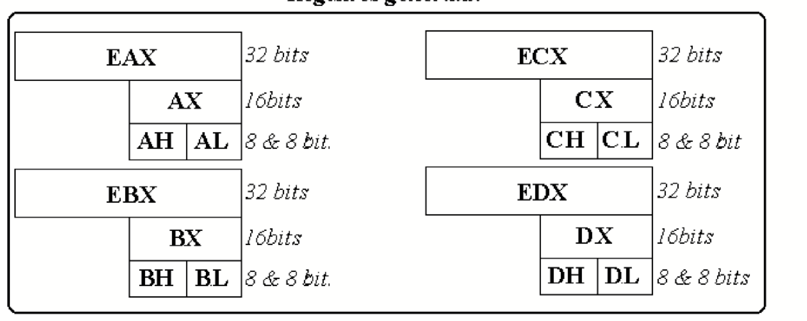
En gérant la mémoire virtuelle (sur le disque dur), windows 3.1 ou windows 95 créent un niveau de cache supplémentaire, le cache de premier niveau est dans votre microprocesseur, le cache de second niveau est sur la carte mère de votre ordinateur, la RAM de votre ordinateur compose le cache de troisième niveau, et l'équivalent de la mémoire est maintenant le disque dur.

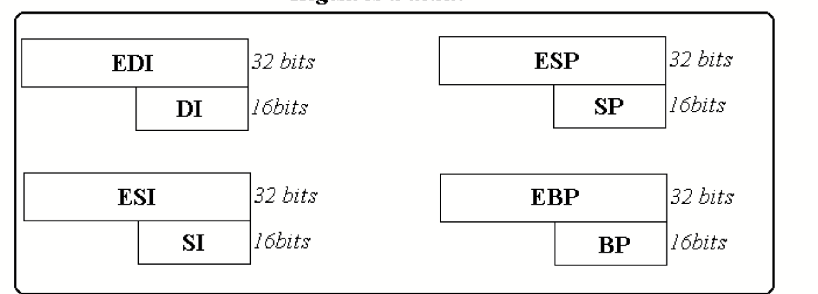
Le microprocesseur cherche la donnée dans son cache, si elle n'y est pas, on cherche dans le cache de second niveau, si elle n'y est pas on la cherche dans la mémoire centrale, et si comble de malchance, on ne la trouve pas, on va donc la chercher sur le disque dur (ce qui prend beaucoup de temps).

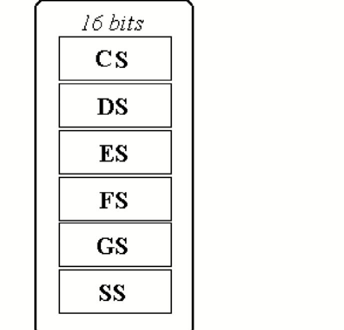
Ne vous affolez pas, au fond ce n'est pas vous le programmeur qui devra gérer tout cela, le microprocesseur s'en charge tout seul. Il existe cependant un intérêt certain à connaître ces mécanismes, on peut se débrouiller, quand on écrit un programme en assembleur (et même en C), pour faire la maximum d'accès mémoire dans le cache et donc le minimum dans la mémoire centrale, ce qui à pour conséquence d'accélérer notablement la vitesse d'exécution, on a alors *optimisé pour le cache,* nous y reviendrons.

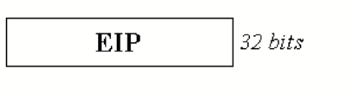
Une imperfection réside dans l'analogie entre les tiroirs et la réalité des accès mémoire: Quand vous accédez à une donnée mémoire, elle reste évidemment présente dans la mémoire, en fait vous faites une sorte de duplication à autre vitesse, une sorte de photocopie de dossiers.

2.4 Les registres









Registre de flag

32 bits

Un registre sert à stocker les données nécessaires à l'exécution d'un programme par le microprocesseur. On n'accède pas à un registre avec une adresse, mais avec son appellation.

Pour les exemples qui vont suivre, je vais vous décrire l'instruction la plus utilisée dans le microprocesseur: MOV

MOV registre1,registre2 a pour effet de copier le contenu du registre2 dans le registre1, le contenu préalable du registre1 étant écrasé. Registre1 et registre2 devant être de même taille.

2.4.1 Les registres généraux

Ils se nomment EAX,EBX,ECX,EDX ce sont des registres de 32 bits, qui servent notamment pour stocker les résultats des opérations arithmétiques.

2.4.1.1 Le registre EAX

On peut accéder aux bits 0 à 7 de ce registre, en utilisant la notation AL (L pour LOW (bas en anglais)).

par exemple: MOV AL,10h

Aura pour effet de placer la valeur 10h dans les bits 0..7 du registre EAX, le reste du registre étant inchangé.

Pour accéder aux bits 8 à 15 de ce registre, on doit utiliser la notation AH (H pour HIGH (haut en anglais))

par exemple: MOV AH,31h

Aura pour effet de placer la valeur 31h dans les bits 8..15 du registre EAX, les autres bits restant inchangés.

On peut accéder aux bits 0 à 15 du registre EAX, en une seule fois, on doit alors utiliser la notation AX.

par exemple: MOV AX,1234h

Aura pour effet de placer la valeur 1234h dans les bits 0..15 du registre EAX.

et enfin pour accéder au registre EAX dans son intégralité, il suffit de l'appeler par son nom EAX.

exemple : MOV EAX,12345678h

Aura pour effet de placer la valeur 12345678h dans le registre EAX.

voyons maintenant un petit programme: ces instructions sont exécutées les unes après les autres.

INSTRUCTION	*CONTENU DE EAX*	*CONTENU DE AX*	*CONTENU DE AH*	*CONTENU DE AL*
MOV EAX,12345678h	12345678h	5678h	56h	78h
MOV AL,10h	12345610h	5610h	56h	10h
MOV AX,0000h	12340000h	0000h	00h	00h
MOV AH,31h	12343100h	3100h	31h	00h
MOV AL,AH	12343131h	3131h	31h	31h

2.4.1.2 Les registres EBX, ECX, EDX

Les registres EBX,ECX,et EDX: conformément au schéma du début du chapitre, se manipulent exactement comme le registre EAX.

2.4.2 Les registres de segments

Ces registres, de 16 bits, servent uniquement, à indiquer l'or d'une écriture ou d'une lecture mémoire, à partir de quel segment on veut lire ou écrire.

Pour donner un exemple, je vais vous présenter un autre aspect de l'instruction MOV.

MOV registre1 , segment: taille [OFFSET]

a pour effet de copier la valeur qui se trouve à l'adresse OFFSET+16*segment de taille 'taille' dans registre1. On dira que l'on à réalisé un adressage direct (l'adresse de la donnée est directement comprise dans l'instruction), dans les cas précédent (pour les premiers exemples), il s'agissait d'adressage immédiat .

taille peut être:

- Un octet on écrit alors BYTE PTR pour taille
- Un mot de 16 bits on écrit alors WORD PTR pour taille
- Un double mot (32 bits) on écrit alors DWORD PTR pour taille.

Exemple:

MOV AL , 200h:BYTE PTR [20h]

a pour effet de copier l'octet qui se trouve à l'adresse 200h:20h, dans le registre AL.

et bien, si on met la valeur 200h dans le registre de segment DS, on pourra réaliser la même chose avec:

MOV AL , DS:BYTE PTR [20h]

Rien ne nous empêche de réaliser une écriture mémoire plutôt qu'une lecture, par exemple:

MOV DS:BYTE PTR [20h], AL

Copie la valeur stockée dans AL, à l'adresse DS:20h soit DS*16+20h.

2.4.3 Rôle des registres de segments

DS 'DATA SEGMENT' est généralement utilisé pour adresser des données.

CS 'CODE SEGMENT' représente le segment à partir duquel se trouvent les instructions du programme (et oui les instructions doivent êtres stockée quelques part en mémoire, tous comme les données, vous reverrez ça dans le dernier chapitre).

ES 'EXTRA SEGMENT ' On peut l'utiliser comme on veux, tous comme **FS** et **GS**.

SS 'STACK SEGMENT ' Comme je ne vous ai pas encore expliqué ce qu'est la pile, j'y reviendrai plus tard.

On ne peut pas mettre directement une valeur immédiate dans un registre de segment, le microprocesseur ne le permet pas:

MOV DS,0200h est incorrect

par contre

MOV AX,0200h
MOV DS,AX est correct.

2.4.4 Les registres d'index

Ce sont des registres de 32 bits, qui peuvent servir à faire un peu n'importe quoi (opérations arithmétiques...), mais qui à l'origine (sur les P.C. XT) servaient à contenir l'offset d'une adresse.

par exemple:

MOV DI,20h

MOV DS:BYTE PTR [DI], AL

est équivalent à

MOV DS:BYTE PTR [20h], AL

Une exception : Dans les registres généraux EBX peut être utilisé comme un registre d'index.

Exemple :

MOV BX,20h
MOV DS:BYTE PTR[BX], AL

est équivalent à l'exemple précédent.

2.4.5 Les différents registres d'index (sur 32 bits):

EDI : 'DESTINATION INDEX', Tire son nom des instructions de copie de chaîne de caractères, ou il pointait sur la destination. Ces instructions ne sont quasiment plus utilisées aujourd'hui. On peut s'en servir pour faire ce que l'on veut à partir du 386.

ESI : 'SOURCE INDEX ' tous comme pour EDI, il servait pour copier des chaînes de caractères, il pointait sur la source. On peut s'en servir pour faire ce que l'on veut à partir du 386.

ESP : 'STACK POINTEUR' Il sert à contenir l'offset de l'adresse de la pile, mieux vaut éviter d'y toucher.

EBP : 'BASE POINTEUR' On en fait, tous comme pour EDI et ESI ce que l'on veut .

Bien entendu, DI,SI,SP,BP représentent les bits de 0 à 15 de ces registres.

2.4.6 Les registres de travail du microprocesseur

2.4.6.1 EIP

Il s'agit d'un registre de 32bits (à partir du 386, avant il s'appelait IP et faisait 16 bits), on ne peut pas y accéder, le microprocesseur s'en sert pour savoir quelle instruction il doit exécuter, ce registre contient l'offset par rapport au registre de segment CS, de l'adresse de la prochaine instruction à exécuter.

2.4.6.2 Registre de flag (32 bits à partir du 386)

On ne peut pas non plus y accéder. Son rôle est de positionner ses bits (à l'état 0 ou 1) selon le résultat arithmétique de la précédente instruction exécutée. Toutes les instructions conditionnelles (instructions qui ne s'exécutent que si une conditions est vérifiée, nous les verrons par la suite..) s'y réfèrent.

BIT	APPELLATION	Rôle
0	CF (Carry flag)	passe à 1, si l'opération arithmétique à engendré une retenue
2	PF (Parity Flag)	passe à 1 si l'opération arithmétique à engendré un résultat avec un nombre pair de bits à 1.
6	ZF (Zero Flag)	Passe à 1 si le résultat de l'opération arithmétique précédente est nul.
7	SF (Sign Flag)	Passe à 1 si le résultat de l'opération arithmétique précédente est négatif
11	OF(Overflow Flag)	Passe à 1, si la somme de deux nombres positif donne par débordement un résultat négatif (voir cour 1)

2.4.7 La pile

La pile est une zone mémoire, dont le programmeur fixe la taille, et qui sert à sauvegarder la valeur d'un registre, tous comme à faire passer des paramètres à un sous programme.

Pour utiliser la pile nous auront 6 instructions:

PUSH --> pour empiler une valeur

POP --> Pour dépiler une valeur

PUSHA --> Pour empiler le contenu de tous les registres du microprocesseur (en 16 bits).

POPA --> Pour dépiler le contenu de tous les registres du microprocesseur (en 16 bits).

PUSHAD --> Pour empiler le contenu de tous les registres du microprocesseur (en 32 bits).

POPAD --> Pour dépiler le contenu de tous les registres du microprocesseur (en 32 bits).

La pile dans les microprocesseurs INTEL, est une pile **LIFO (**Last In First Out) ce qui signifie que la première valeur dépilée, sera la dernière que l'on aura empilé.

Exemple:

```
MOV   AX,0010h
MOV   BX,1234h

PUSH  AX
PUSH  BX

POP   AX            ==> AX contient 1234h
POP   BX            ==> BX contient 0010h
```

cet exemple à permis d'échanger le contenu de deux registres, en se servant de la pile comme tampon intermédiaire.

2.5 *Assembleur et langage machine*

Dans des discutions passionnées sur les compétences de chacun, il est rare que quelqu'un ne sorte pas l'idiotie suivante:

Le langage machine c'est plus rapide que l'assembleur !

Ou pire encore :

L'assembleur, c'est génial, c'est plus rapide que le langage machine !

Rassurer vous, si vous avez à faire à ce genre de personnes, ne vous sentez pas ignorant, il s'agit de personnes qui ne savent pas de quoi elles parlent, et qui se permettent de porter des jugements.

Le langage machine c'est exactement la même chose que l'assembleur, seule l'apparence diffère, je m'explique.

Si vous voulez mettre la valeur FFFFh dans AX vous taperez

MOV AX,FFFFh

et votre assembleur transformera ça en

B8FFFF

soit en binaire %1011 1000 1111 1111 1111 1111.

quand votre microprocesseur rencontrera la valeur binaire %10111000, il saura que il s'agit de l'instruction MOV AX,? et que vous allez lui fournir à la suite une valeur immédiate sur 16 bits qu'il devra mettre dans AX.

Si vous aviez tapé directement les bits un à un, le résultat aurait été exactement le même, vous auriez simplement beaucoup souffert pour rien, vous auriez alors programmé en langage machine.

L'assembleur, se limite en fait à directement transcrire en code machine votre programme assembleur. L'assembleur ne modifiera en rien vos initiatives , la vitesse d'exécution est donc exactement la même, que le programme ait été programmé en langage machine bit par bit , ou en assembleur avec turbo assembleur (ou watcom assembleur).

Si par contre, vous programmez en basic ou en langage C, vous ne saurez pas ce que le compilateur fera de votre programme quand il le transformera en un programme machine directement compréhensible par le microprocesseur.

Ces petites précision, devraient vous permettre de comprendre maintenant pourquoi les instructions ont une adresse, elles sont pointées par CS:EIP, et c'est le microprocesseur qui les interprétera comme données ou instructions selon le contexte. Vous verrez, que, quand vous programmerez si par mégarde vous allez continuer l'exécution de votre programme dans des données, le microprocesseur se retrouvera alors avec des instructions incohérentes, et plantera assez vite.

Chapitre 3

3 La mémoire en détail

3.1 Segmentation de la mémoire

La taille maximum d'un segment est de 64 Ko.
Pour qu'un programme s'exécute, le segment de code et le segment de pile doivent exister.

3.2 Les emplacements réservés

Certaines zones de la mémoire sont réservées à des usages particuliers :

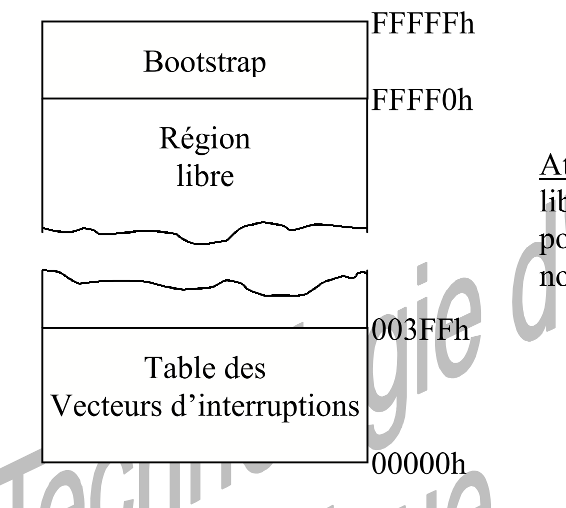
a) De FFFF0h à FFFFFh, réservé au bootstrap, c'est à dire à l'initialisation du système (vecteur RESET).

   Note : on trouve en FFFF0h EA E07D F000 ≡ JMP F000 :E07D qui correspond au point d'entrée du BIOS

b) De 0 à 3FFh, réservée aux vecteurs d'interruptions. Chaque vecteur nécessite 4 octets (base + offset).
   Le microprocesseur est capable de reconnaître jusqu'à 256 niveaux d'interruption. Lorsqu'une interruption est générée, le microprocesseur identifie le demandeur et vient chercher dans la table des vecteurs, l'adresse de la routine de gestion du demandeur.

Routine de gestion d'un périphérique
CSHCSL:IPHIPL
IRET
Pointe la 1ère instruction à exécuter
0000B
00007
00003
00000
CSH
CSL
IPH
IPL
Vecteur 2
Vecteur 1
Vecteur 0

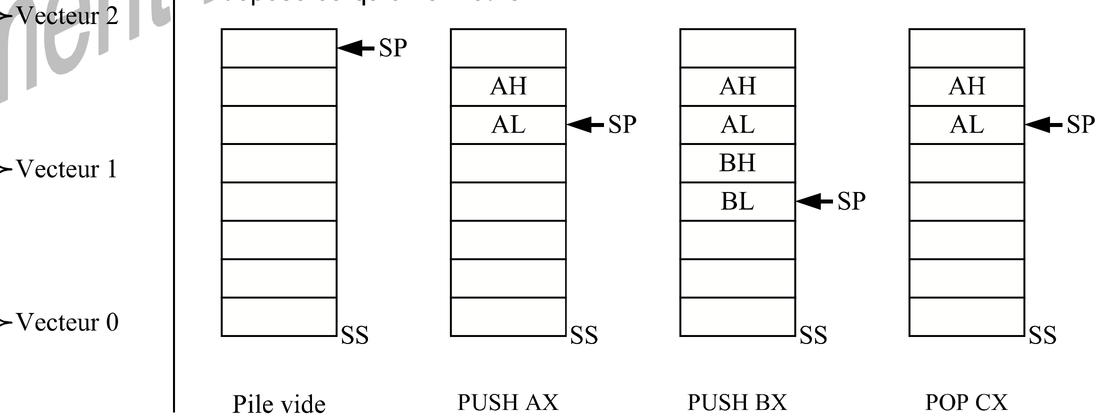
En résumé :



Attention : La région libre est aussi utilisée pour le stockage du noyau

3.3 La pile

C'est une région de la RAM où l'on va stocker ou retirer des informations. L'utilisation de la pile est très fréquente pour la sauvegarde temporaire du contenu des registres et pour le passage de paramètres lorsqu'un langage de haut niveau fait appel à une routine en assembleur.
La taille de la pile varie en fonction de la quantité d'informations qu'on y dépose ou qu'on en retire.



Le pointeur de pile (SP) contient au départ l'adresse supérieure de la région qu'elle peut occuper.

- L'opération consistant à déposer des informations dans la pile (PUSH) effectue une décrémentation du pointeur SP et range ensuite la donnée à l'adresse SS :SP et SS :SP+1 (dans le cas de l'exemple puisqu'il s'agit de données sur 16 bits),

- L'opération consistant à retirer des informations dans la pile (POP) extrait du sommet de la pile le premier mot et incrémente SP (2 positions) pour pointer sur un nouveau mot.

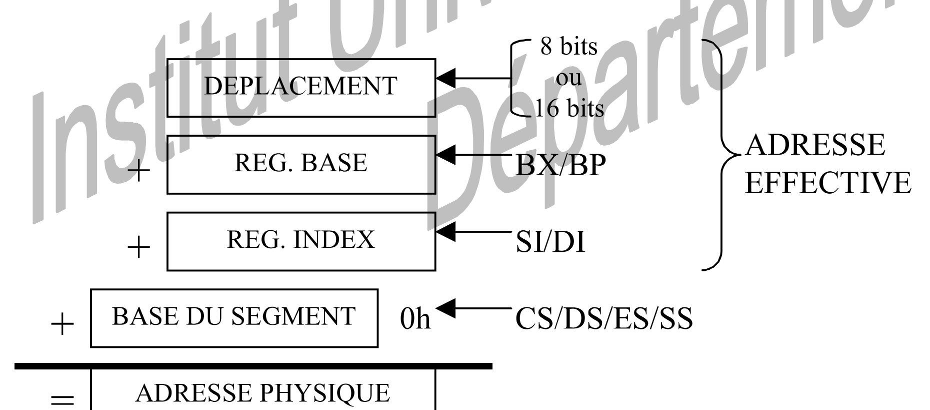
3.4 Les modes d'adressage

Domaine physique d'adressage direct : 1 Mo

Domaine logique d'adressage direct : 4 * 64 Ko

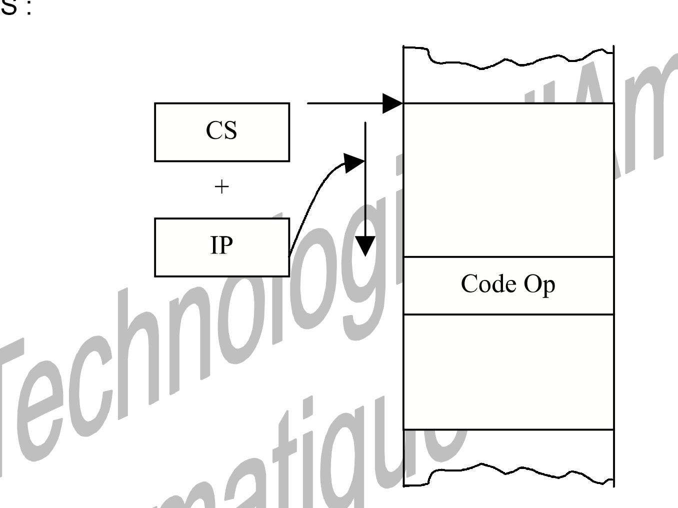
Modes d'adressages :
- direct registres
- direct mémoire
- indirect via registre de base
- avec ou sans indexation par registre d'index

Calcul d'une adresse :



3.4.1 Accès aux instructions

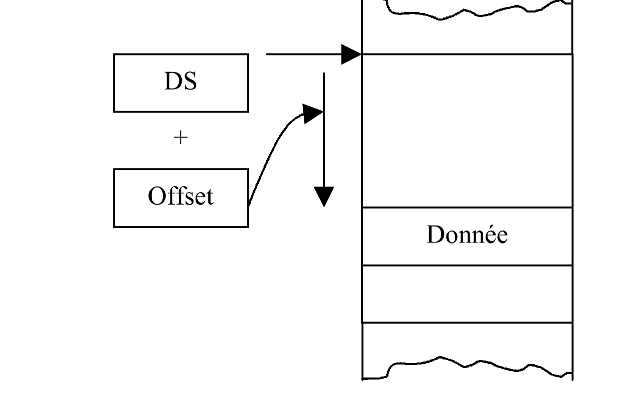
L'accès aux instructions fait implicitement appel à IP basé par rapport à CS :



3.4.2 Accès aux données

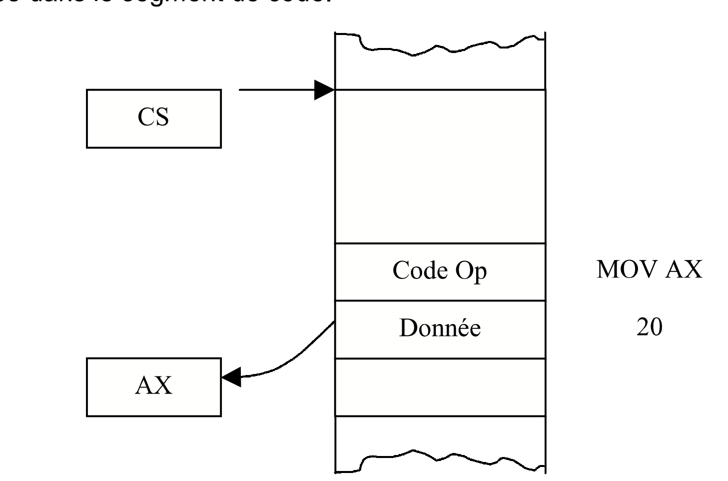
Lorsqu'un segment logique est ouvert dans un programme source, a l'assemblage, un compteur lui est alloué.

Ainsi l'assembleur fait correspondre à chaque étiquette ou nom de variable une valeur de compteur qui est simplement l'offset du symbole.



3.4.3 Adressage immédiat

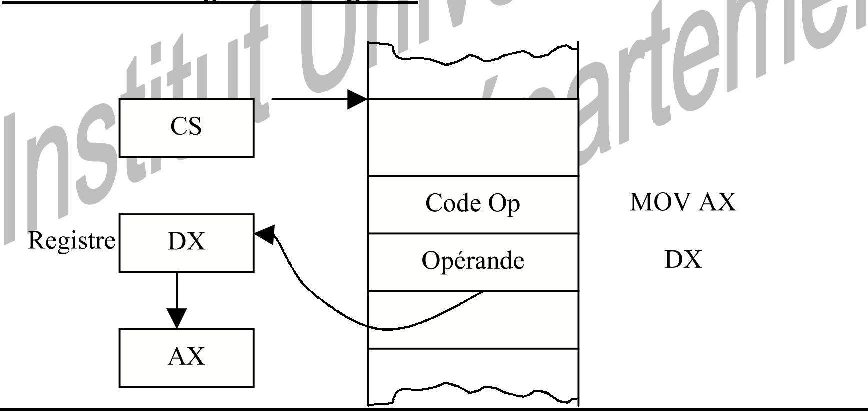
La donnée est spécifiée immédiatement après l'instruction. Elle est donc située dans le segment de code.



Quelques exemples :

MOV AX, 20
MOV CL, 8
MOV BX, 15h

3.4.4 Adressage direct registre



Quelques exemples :

MOV AX, DX
MOV CH, AL

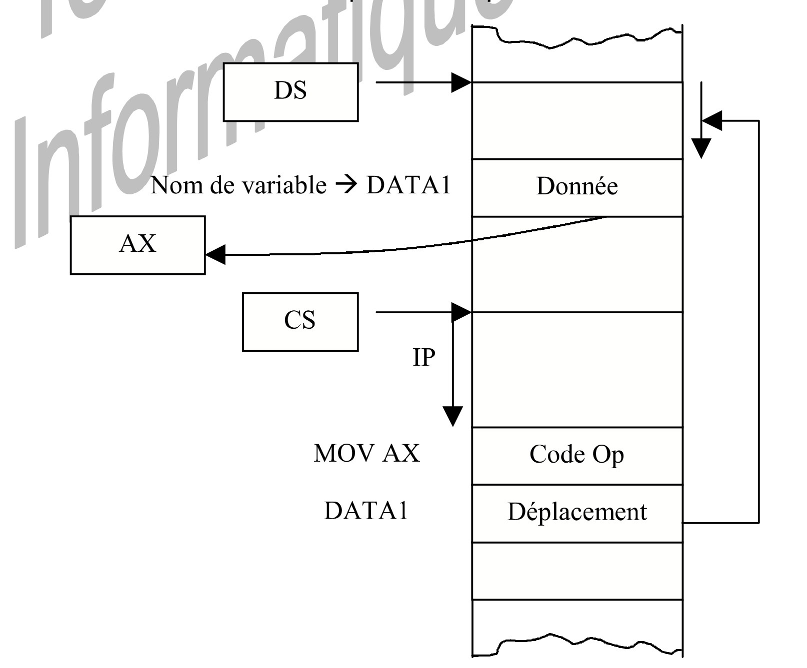
Attention : les données doivent être de même type, l'instruction suivante est mauvaise : MOV CX, BL.

3.4.5 Adressage direct mémoire (déplacement)

L'assembleur traduit les étiquettes en offsets.
Les données sont implicitement basées par DS.
Il est possible de spécifier une autre base :

MOV AX, ES :DATA1

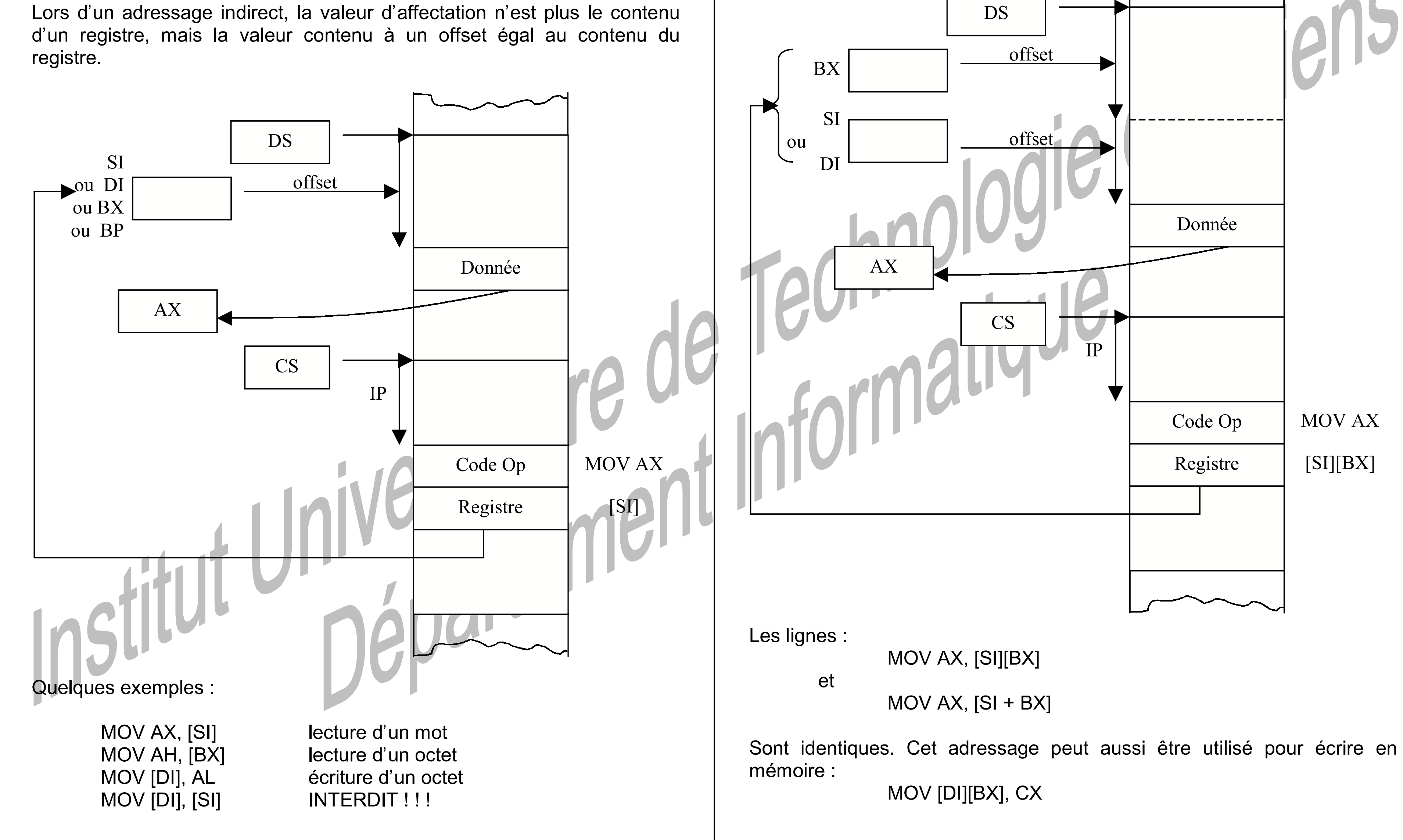
DATA1 est alors basé par rapport à ES.
⇒ l'instruction nécessitera plus de temps lors de l'exécution.



3.4.6 Adressage indirect mémoire

3.4.6.1 Base ou index

Lors d'un adressage indirect, la valeur d'affectation n'est plus le contenu d'un registre, mais la valeur contenu à un offset égal au contenu du registre.



Quelques exemples :

MOV AX, [SI]	lecture d'un mot
MOV AH, [BX]	lecture d'un octet
MOV [DI], AL	écriture d'un octet
MOV [DI], [SI]	INTERDIT ! ! !

3.4.6.2 Base et index

Les lignes :

MOV AX, [SI][BX]

et

MOV AX, [SI + BX]

Sont identiques. Cet adressage peut aussi être utilisé pour écrire en mémoire :

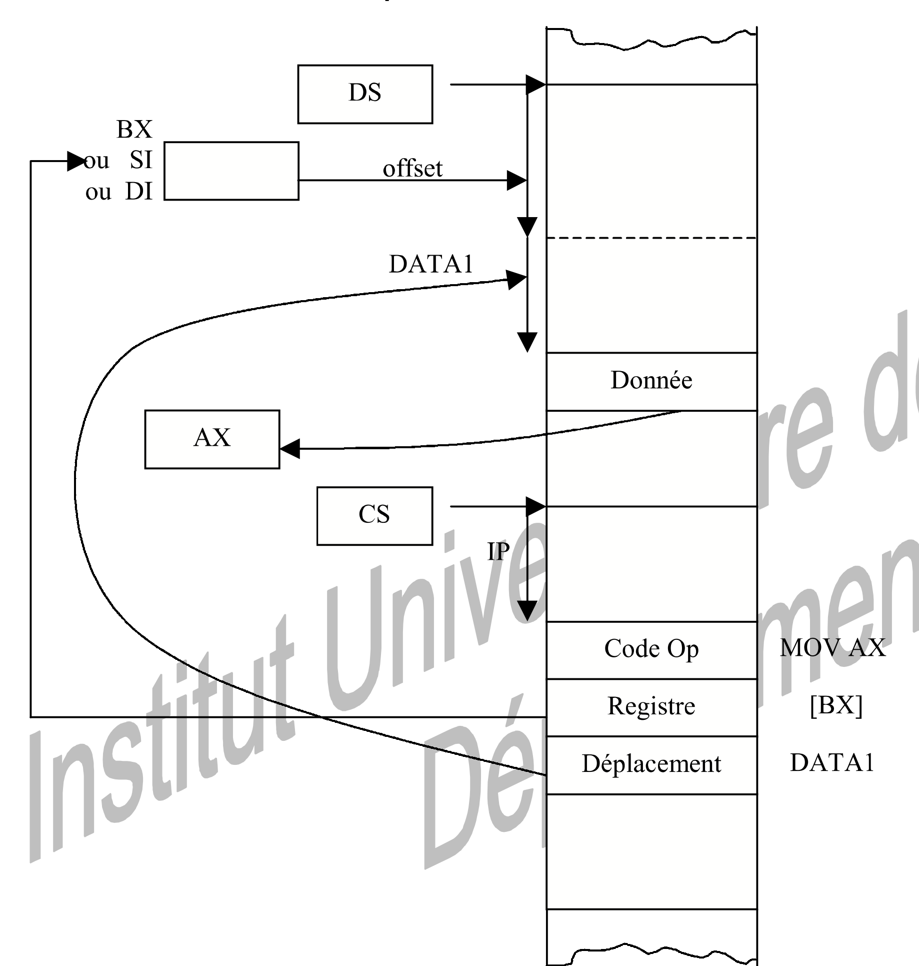
MOV [DI][BX], CX

Par contre, la ligne suivante est fausse :

```
MOV DX, [SI] [DI]
```

Car l'utilisation simultané de deux index est interdite pour un adressage.

3.4.6.3 Base ou index + déplacement



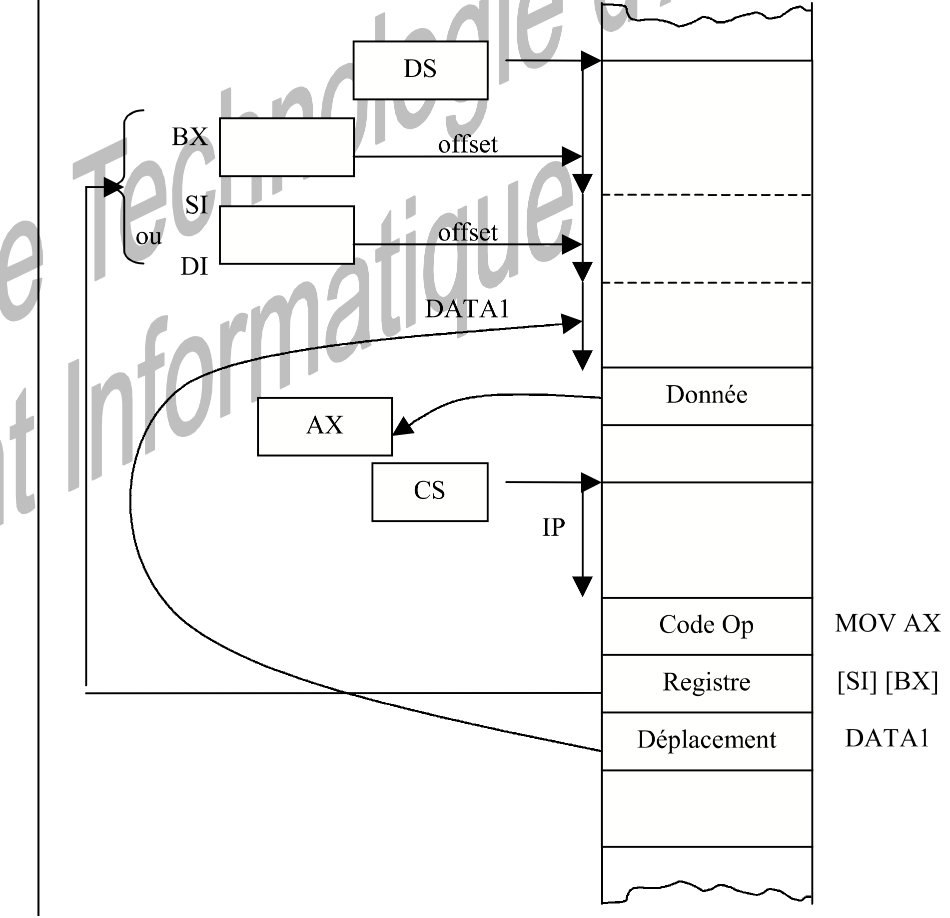
Il est possible d'utiliser ce mode d'adressage pour la lecture ou l'écriture :

```
MOV AX, [BX] DATA1
MOV [SI] DATA2, CX
```

Il est possible aussi d'ajouter un déplacer supplémentaire à l'aide du valeur immédiate : MOV AX, [DI] DATA3(2)

Ici, on effectue un déplacement supplémentaire de deux octets.

3.4.6.4 Base + index + déplacement



En résumé :

Exemple d'instruction	Mode d'adressage
MOV AX, 5	Immédiat
MOV AX, BX	Registre
MOV AX, DATA1	Direct mémoire
MOV AX, [BX]	Indirect base
MOV AX, [SI]	Indirect index
MOV AX, [BX + 5]	Indirect base + déplacement
MOV AX, [SI + 5]	Indirect index + déplacement
MOV AX, [BX][SI] MOV AX, [BX + DI]	Indirect base + index
MOV AX, [BX][DI] 5 MOV AX, [BX + SI + 5]	Indirect base + index + déplacement

Chapitre 4

4 LE JEU D'INSTRUCTIONS

4.1 Généralité

Une instruction en assembleur se traduit en une suite d'octets représentant le langage machine équivalent.
Cette suite d'octets a une longueur qui varie selon les instructions.

Comme nous l'avons déjà vu, le microprocesseur possède un registre particulier qu'est le registre de d'indicateurs (flag ou drapeau). Ce registre est en fait composé de bits qui ont des significations différentes et qui sont modifiés (positionnés) selon l'exécution de certaines instructions.

Positionnement des indicateurs après exécution des instructions :

	A	C	O	P	S	Z
Arithmétique	X	X	X	X	X	X
INC / DEC	X	?	X	X	X	X
Logique	?	O	O	X	X	X
Décalage	?	X	X	X	X	X
Rotation	?	X	X	?	?	?

X → Positionné selon le résultat
O → Mis à zéro
? → Non modifié sauf si rapport avec le résultat

Positionnement des indicateurs à l'aide d'instructions :

	1	0	Complément
Carry	STC	CLC	CMC
Direction	STD	CLD	
Interupt Enable	STI	CLI	

- CLI interdit les interruptions
- STI autorise les interruptions

4.2 Les instructions de base

4.2.1 L'identificateur

Une instruction peut être précédée d'un identificateur (de forme générale Ident:) qui représente l'adresse de stockage de cette instruction. On appelle cet identificateur une étiquette (label en anglais).

```
Debut: MOV AX, 00h
INC AX
JMP Debut
```

4.2.2 MOV

L'instruction de transfert de données se fait sur 8, 16 ou sur 32 bits.

Notes : reg = registre 8, 16 ou 32 bits
mem = adresse (ou identificateur)
imm = constante (valeur immédiate)

```
MOV reg, imm
MOV mem, imm
MOV reg, mem
MOV mem, reg
MOV reg, reg
```

Déplace l'opérande de droite dans l'opérande de gauche.
MOV AH, 00h
N'affecte pas les indicateurs.
Si un opérande est de type indéfini,

```
MOV [AX], 00h ->   MOV BYTE PTR [AX], 00h
                ou
                   MOV WORD PTR [AX], 00h
```

4.3 Les instructions de pile

Une pile est une zone mémoire servant à stocker temporairement des valeurs. On ne peut stocker qu'une information à la fois et l'élément dépilé à un moment donné est celui qui a été empilé en dernier : c'est la structure LIFO (Last In, First Out). Les opérations ne se font que sur 16 bits.

La pile commence au segment SS et elle fini dans le même segment mais à l'offset SP. A notez que cette pile est remplie à l'envers : le premier élément est situé à une adresse plus haute que le dernier élément.

4.3.1 PUSH

Empile l'opérande. SP est décrémenté de 2.

```
PUSH reg
PUSH mem
```

4.3.2 POP

Dépile un élément et le copie dans l'opérande.
SP est incrémenté de 2.

```
POP reg
POP mem
```

Note : MOV DS,ES est interdit => PUSH ES et POP DS

4.3.3 PUSHA (PUSH All)

Empile tous les registres. Ne convient pas pour les registre 32 bits.

```
PUSHA
```

4.3.4 POPA (POP All) :

Dépile tous les registres. Ne convient pas pour les registres 32 bits.

```
POPA
```

4.3.5 PUSHAD (PUSH All) :

Même principe que PUSHA, mais empile aussi les registres 32 bits.

```
PUSHAD
```

4.3.6 POPAD (POP All)

Même principe que POPA, mais dépile aussi les registres 32 bits.

```
POPAD
```

4.4 Les instructions sur le registre indicateur

4.4.1 PUSHF (PUSH Flag)

Permet de placer au sommet de la pile le contenu du registre indicateurs.

```
PUSHF
```

4.4.2 POPF (POP Flag)

Recopie le sommet de la pile dans le registre d'état.

```
POPF
```

4.4.3 LAHF (Load AH from Flag)

Cette instruction assure le transfert des indicateurs SF, ZF, AF et CF dans AH.

LAHF → AH :

SF	ZF		AF		PF		CF

4.4.4 SAHF (Store AH into Flag)

Cette instruction effectue le transfert de AH vers les indicateurs.

```
SAHF
```

4.5 Les instructions de transferts accumulateur

4.5.1 IN (INput from port):

Cette instruction transfert les données contenu dans le port spécifié dans AL,AX ou EAX.Si le numéro du port est inférieur à 256, alors l'adresse peu être mis tel que, sinon, il faut mettre l'adresse dans le registre DX.

```
        IN AL, 42h
    sinon
        MOV DX, 123h
        IN AX, DX
```

4.5.2 OUT (OUTput to port)

Comme l'instruction IN, OUT permet la lecture d'un port spécifié dans l'accumulateur AL, AX ou EAX. Si le numéro du port est inférieur à 256, alors l'adresse peu être mis tel que, sinon, il faut mettre l'adresse dans le registre DX.

```
        OUT 8, AL
    sinon
        MOV DX, 126h
        OUT DX, AX
```

4.5.3 XLAT

Cette instruction traduit un octet au moyen d'une table de transcodage.
→ Le pointeur de début de table se trouve dans BX,
→ L'accumulateur AL est utilisé comme index dans la table. L'opérande ainsi adressé est transféré dans AL.

```
    MOV BX, offset TABLE
    MOV AL, 2
    XLAT
```

Remarques :

1) La table comportera au plus 256 valeurs,
2) L'utilisation de cette instruction est fréquente pour la ré affectation de caractères, et les transcodages binaire ⇔ ASCII.

4.6 Les instructions arithmétiques

4.6.1 L'addition

4.6.1.1 ADD

Cette instruction effectue une addition, le résultat est placé dans le premier opérande.

ADD reg,imm ADD mem,imm ADD reg,mem ADD mem,reg ADD reg,reg	ADD BX, 1 ADD AX, [BX + 2]

Flags modifiés : AF - CF - OF- SF - PF - ZF
Exemple :

```
MOV AX, 45h
ADD AX, 7

AX vaut maintenant 4Ch
```

4.6.1.2 ADC (Add with Carry)

Cette instruction effectue une addition(résultat placé dans le premier opérande), mais le résultat varie en fonction de CF, si CF=0:le résultat est le résultat de l'addition, si CF=1 le résultat est incrémenté de 1.

Flags modifiés : AF - CF - OF- SF - PF - ZF
Exemple :

```
CF=1
AX = 45h

ADC AX, 2

AX vaut donc maintenant 48h
Si CF= 0, alors AX vaudrait 47h
```

4.6.1.3 INC (INCrement)

Cette instruction est en fait l'équivalant de "ADD ?,1 ", cette instruction incrémente de1 l'opérande.
Flags modifiés : AF - OF - PF - SF - ZF
Exemple:

```
AX contient 18h ( 24 en décimal)

INC AX

AX contient 19h
```

4.6.2 La soustraction

4.6.2.1 SUB (SUBstract)

Cette instruction soustrait le deuxième opérande au premier.

Flags modifiés : AF - OF - PF - SF - ZF
Exemple :

```
AX contient 18h ( 24 en décimal)

SUB AX, 5

AX=13
```

4.6.2.2 SBB (SUBstract with Below)

Cette instruction est en fait un dérivé de l'instruction SUB, elle fait aussi une soustraction, mais décrémente en plus de1 le résultat, si l'indicateur carry (CF) est égal à 1.

Flags modifiés : AF - OF - PF - SF - ZF
Exemple :

```
AX contient 18h ( 24 en décimal)
CF=1

SBB AX, 5

AX=12
```

4.6.2.3 DEC (DECrement)

Cette instruction fait l'équivalant de "SUB ?,1 ", cette instruction décrémente donc de1 l'opérande.

Flags modifiés : AF - OF - PF - SF - ZF
Exemple :

```
AX contient 18h ( 24 en décimal)

DEC AX

AX contient 17h
```

4.6.3 La multiplication

4.6.3.1 MUL (MULtiply)

Cette instruction sert à multiplier un nombre Non Signé par un nombre source. Il y a 3 cas différents :

- Si l'opérande source à une taille de un octet, AL est multiplié par la source, et le résultat dans AX,
- Si l'opérande source est un mot, le registre AX est multiplié par celui-ci, et le résultat placé dans le registre 32 bits : DX+AX,
- Si l'opérande source est un double, le registre EAX est multiplié par l'opérande source, le résultat placé dans le registre 64 bits : EDX+EAX.

Flags modifiés : CF - OF
Exemple :

```
AX contient 9
BL contient 3

MUL BL

AX contient 27
```

4.6.3.2 IMUL (signed Integer MULtiply)

Cette instruction a la même fonction que l'instruction MUL ci-dessus mais elle supporte les nombres signés.

Flags modifiés : CF - OF
Exemple :
Voir MUL

4.6.4 La division

4.6.4.1 DIV (DIVise)

Cette instruction sert à diviser un nombre Non Signé par un nombre source; il y a 3 cas différents :

- Si l'opérande source a une taille de un octet, AX est divisé par celui-ci, le quotient dans AL et le reste dans AH,
- Si l'opérande source est un mot, le nombre 32 bits (DX+AX) est divisé par celui-ci, le quotient dans AX et le reste dans DX,
- Si l'opérande source est un double, le nombre 64 bits (EDX+EAX) est divisé par l'opérande source, le quotient dans EAX et le reste dans EDX.

Flags modifiés : Aucun
Exemple :

```
AX contient 18
BL contient 3

DIV BL

AX contient 9
```

4.6.4.2 IDIV (signed Integer DIVision)

Cette instruction a la même fonction que l'instruction DIV ci-dessus mais elle supporte les nombres signés.

Flags modifiés : Aucun
Exemple :
Voir DIV

4.6.4.3 CMP (CoMPare)

Cette instruction compare deux opérandes; pour cela cette instruction soustrait les deux opérandes. Le résultat de la soustraction n'est pas Sauvé, seuls les indicateurs sont modifiés.

Flags modifiés : AF - CF - OF - PF - SF - ZF
Exemple :

```
CMP AX, 7
JNZ Suite        ;Si AX =/= 7 va à suite

...
Suite: ...
```

4.6.5 Ajustement de données

4.6.5.1 AAA (Ascii Adjust for Addition)

Il faut utiliser cette instruction après avoir effectué une addition, cette instruction va corriger le contenu du registre AL, pour que l'addition soit juste.

Flags modifiés : AF - CF
Exemple :

AX contient la valeur 19, on veut addition 3 au registre AX:

AX = 0109h

ADD AX, 3

AX devient 010Ch, or on veut additionner 3 au nombre ASCII 19; on utilise donc AAA:

AAA

AX va être ajusté et va maintenant contenir 0202h

4.6.5.2 AAS (Ascii Adjust for Subtraction)

Cette instruction corrige le résultat après une soustraction de nombres non compactés.

Flags modifiés : AF - CF
Exemple :

Voir AAA

4.6.5.3 AAM (Ascii Adjust for Multiplication)

Cette instruction corrige le résultat après une multiplication de nombres non compactés.

Flags modifiés : PF - SF - ZF
Exemple :

AL contient la valeur 3, on veut multiplier AL par 9
AL=0003h
BL=0009h

MUL BL

AX contient maintenant 001Bh . (27 en décimal)

AAM

AL contient 0207h

4.6.5.4 AAD (Ascii Adjust for Division)

Cette instruction doit être utilisée avant de faire une division de nombre décimaux non compactés dans un registre. Cette procédure va en fait multiplier le contenu de AH par 10, et additionner le résultat à AL; AH est remis ensuite à 0.

Flags modifiés : SF - ZF - PF
Exemple :

AX contient la valeur 19.

AX = 0109h

AAD

AX contient maintenant 0013h.

4.6.6 Conversion de type

4.6.6.1 CBW (Convert Byte to Word)

Cette instruction convertit le registre AL, en registre AX. Cette instruction travaille aussi avec des nombre signés.

Flags modifiés : Aucun

<u>Exemples</u> :

AL=1Eh (soit 30)
AH=? le contenu de AH, n'est pas important.

CBW

AX vaut maintenant 001Eh (30)

Si AL = E2h (-30)

CBW

AX vaut maintenant FFE2h (-30)

4.6.6.2 CWD (Convert Word to Doubleword)

Cette instruction est du même type que CBW, mais convertit un Mot en Double Mot. Le résultat est placé dans AX et DX. Travaille aussi avec des nombres signés.
<u>Flags modifiés</u> : Aucun
<u>Exemples</u> :

AX = 1234h (4460en décimal)
DX= ?

CWD

AX=1234h
DX=0000h

<u>De même</u> :
AX= EDCCh(-4460 décimal)

CWD

AX inchangé
DX=FFFFh

4.6.6.3 CWDE (Convert Word to Extended Doubleword)

Cette instruction est presque la même que celle ci-dessus, mais le résultat est placé dans EAX au lieu d'être placé dans AX et DX.

<u>Flags modifiés</u> : Aucun
<u>Exemples</u> :

EAX=????1324h

CWDE

EAX=00001234h

<u>De même</u> :
EAX=????EDCCh(-4460 décimal)

CWDE

EAX=FFFFEDCCh

4.6.6.4 CDQ (Convert Doubleword to Quadword)

Cette instruction convertit le registre EAX,en registre 64 bits (EAX+EDX). Cette instruction travaille aussi avec des nombres signés.

<u>Flags modifiés</u> : Aucun
<u>Exemples</u> :

EAX=12345678h (soit 305419896)

CDQ

EAX vaut maintenant 12345678h
EDX vaut maintenant 00000000h

Si EAX = EDCBA988h (-305419896)

CDQ

EAX est inchangé
EDX = FFFFFFFFh

4.6.6.5 NEG (NEGation):

Cette instruction va tout simplement inverser le contenu d'un registre ou d'une adresse mémoire. Pour cela, elle inverse d'abord tous les bit, et ajoute 1.

Flags modifiés : AF - OF - PF - SF - ZF
Exemple :

AX contient 1234h (soit 4660)
AX=0001 0010 0011 0100

NEG AX

AX=1110 1101 1100 1011 +0000 0000 0000 0001
AX=1110 1101 1100 1100
AX=FFFFEDCCh Soit (-4660)

4.7 Les instructions logiques

4.7.1 AND (ET logique)

Cette instruction est l'équivalant d'une porte ET logique, elle réalise un ET entre 2 opérandes. Le résultat est placé dans le premier opérande.

Flags modifiés : CF - OF - PF - SF - ZF
Exemple :

On effectue un ET entre AH (9Dh) et BH (6Dh)

AH=1001 1101
BH=0110 1101

AND AH, BH

AH=0000 1101 (0Dh)

4.7.2 OR (OU logique)

Encore une autre porte logique, cette porte effectue un OU entre deux opérandes.

Flags modifiés : CF - OF - PF - SF - ZF
Exemple :

On effectue un OU entre AH (9Dh) et BH (6Dh)

AH=1001 1101
BH=0110 1101

OR AH, BH

AH=1111 1101 (FDh)

4.7.3 XOR (OU eXclusif)

Cette instruction effectue un OU exclusif.

Flags modifiés : CF - OF - PF - SF - ZF

Exemple :

On effectue un XOR entre AH (9Dh) et BH (6Dh)

AH=1001 1101
BH=0110 1101

XOR AH, BH

AH=1111 0000 (F0h)

4.7.4 TEST (TEST for bit pattern)

Cette instruction va effectuer un ET logique, mais le résultat ne sera pas gardé; seul les indicateurs seront modifiés

Flags modifiés : CF - OF - PF - SF - ZF

4.7.5 NOT

C'est l'équivalent de la porte logique NON, le principe de cette porte est simple, inverser tous les bits.

Flags modifiés : Aucun
Exemple :

```
AX=1001 1101 1110 0110 (9DE6h)

NOT AX

AX= 0110 0010 0001 1001 (6219h)
```

4.8 *Les instructions de décalages et de rotations*

En mode immédiat, seuls les décalages et rotations de 1 bit peuvent se réaliser. Pour réaliser des décalages ou rotations de plusieurs bits, il faut explicitement utiliser le registre CL.

4.8.1 SHR (SHift Right)

Décale à droite de x positions, x contenu dans le deuxième opérande, les bits entrant sont mis à 0 (on ne fait pas attention au signe, contrairement à SAR), les bits sortant sont placés successivement dans CF, sans conservation, celui-ci prend donc la valeur du dernier bit sorti.

Flags modifiés : CF - OF - PF - SF - ZF
Exemple :

On va décaler le registre AH de 5 positions.

```
AH =1011 0111

SHR AH, 5

AH=00000101
```

4.8.2 SHL (SHift Left)

Effectue un décalage des bits du premier opérande, les bits sortant sont insérés dans CF, les bits entrant sont mis à 0.

Flags modifiés : CF - OF - PF - SF - ZF
Exemple :

On va décaler le registre AH de 5 positions.

```
AH =1011 0111

SHL AH, 5

AH=1110 0000
```

4.8.3 SAR (Shift Arithmetic Right)

Cette instruction effectue un décalage vers la droite, le nombre de positions à décaler est inscrit dans le second opérande. Le bit sortant est mis dans CF, et le bit rentrant varie en fonction du signe du nombre au départ; si au départ le nombre était positif, tous les bit entrant seront de 0, sinon, l'inverse. Comme le carry (CF) prend successivement les valeurs des bits sortant, après cette instruction CF est égal au dernier bit sorti.

Flags modifiés : CF - OF - PF - SF - ZF
Exemple :

On va décaler le registre AH de 5 positions vers la droite.

```
AH = 0101 1100 (92d)

SAR AH, 4

AH= 0000 0101
CF=1
```

4.8.4 SAL (Shift Arithmetic Left)

Cette instruction est identique à SHL.

4.8.5 ROR (ROtate Right)

A peu près le même style que les instructions précédentes, les bits sont décalés vers la droite, mais, cette fois, le bit sortant est injecté à dans CF et dans le bit de poids fort.

Flags modifiés : CF - OF
Exemple :

On va décaler le registre AH de 5 positions vers la droite.
AH =1011 0111

ROR AH, 5

AH= 0111 1101

4.8.6 ROL (ROtate Left)

Idem ROR, mais décale sur la gauche.

Flags modifiés : CF - OF
Exemple :

On va décaler le registre AH de 5 positions vers la gauche.
AH =1011 0111

ROL AH, 5

AH=1111 0110

4.8.7 RCR (Rotate trought Cary Right)

Cette instruction agit sur les bits de l'opérande, elle les décale de x vers la droite. Le bit contenu dans CF prend la position du bit le plus fort, puis CF prend la valeur du bit sortant. X est la valeur contenu dans le deuxième opérande.

Flags modifiés : CF - OF
Exemple :

On va décaler le registre AH de 5 positions.
On sait que CF=0
AH =1011 0111

RCR AH, 5

AH=01110101
CF=1

4.8.8 RCL (Rotate trought Cary Left)

Idem RCR, mais déplace les bits vers la gauche.

Flags modifiés : CF - OF
Exemple :

On va décaler le registre AH de 5 positions.
On sait que CF=1
AH =1011 0111

RCL AH, 5

AH=1111 1011
CF=0

4.9 Ruptures de séquence

4.9.1 Les branchements inconditionnels

4.9.1.1 JMP (JUmP)

Cette instruction effectue un saut sans condition. Le saut peut être sur un label, une adresse mémoire, ou un registre (adressage direct ou basé).

Flags modifiés : Aucun
Exemple :

```
JMP label1
XORAX,AX
Label1:
```

XOR est sauté.

4.9.1.2 INT (INTerrupt)

C'est assez simple, une interruption est un genre de sous programme contenu dans le BIOS ou dans votre système d'exploitation. Les interruptions vous facilite la tache, mais des fois, il vaut mieux éviter de les utiliser.
Je m'explique : Il existe par exemple une interruption qui permet d'afficher un pixel à l'écran, en 320*200, le problème, c'est que ca va très lentement,

surtout si on l'appelle 64000 fois!!! Pour ce cas, il existe donc un autre moyen, chaque pixel de l'écran est représenté dans une partie de la mémoire, et il suffit donc de mettre la couleur du pixel à l'adresse ou l'on veut pour afficher un pixel.

La plupart du temps, il vous faudra modifier les registres pour lancer un interruption.

Flags modifiés : TF - IF
Exemple :

```
MOV AH, 09h
INT 21h
```

Fonction MS-DOS; Ecrit un message.

```
MOV AH, 00h
MOV AL, 13h
INT 10h
```

Fonction BIOS; Charge le mode VGA 256 couleur 320*200.

4.9.2 Les branchements conditionnels

Ces instructions permettent de sauter "une partie d'un programme" si une condition est vérifiée. Elle est très pratique, on peut par exemple écrire de petit programmes du type basic "if...then ....else". Toutefois si les conditions ne sont pas vérifiées, le programme saute cette instruction et suit son cours normal.
Toutes ces instructions ont la forme générale suivante : J… *indent*, où « J… » est le mnémonique de l'instruction et « *ident* » est un label qui doit se trouver dans un intervalle de –127 à 128 autour de l'instruction.

4.9.2.1 Les tests d'indicateurs

Instruction	Condition	Définition
JZ	ZF = 1	Jump if Zero
JE		Jump if Equal
JNZ	ZF = 0	Jump if Not Zero
JNE		Jump if Not Equal
JC	CF = 1	Jump if Carry
JNC	CF = 0	Jump if Not Carry
JS	SF = 1	Jump if Sign
JNS	SF = 0	Jump if Not Sign
JO	OF = 1	Jump if Overflow
JNO	OF = 0	Jump if Not Overflow

4.9.2.2 Les test de nombres non signés

Instruction	Condition	Indicateurs	Définition
JA	A > B	CF = 0 et ZF = 0	Jump if Above
JNBE			Jump if Not Below or Equal
JAE	A >= B	CF = 0	Jump if Above or Equal
JNB			Jump if Not Below
JNC			Jump if Not Carry
JBE	A <= B	(CF = 1 et ZF = 1) ou (CF <> ZF)	Jump if Below or Equal
JNA			Jump if Not Above
JB	A < B	CF = 1	Jump if Below
JC			Jump if Carry
JNAE			Jump if Not Above or Equal
JE	A = B	ZF = 1	Jump if Equal
JZ			Jump if Zero
JNE	A <> B	ZF = 0	Jump if Not Equal
JNZ			Jump if Not Zero

Exemple :

```
    MOV AH, 0FFh
    CMP AH, 0           ;compare FFh avec 0
    JB suite            ;test NON signé (saut si AH < 0)
    …
    …
Suite :
    …
    …
```

L'instruction JB **ne provoquera pas le branchement** au label SUITE car 255 est supérieur à 0 car comme le test est non signé, FFh vaut 255.

4.9.2.3 Les tests de nombres signés

Instruction	Condition	Indicateurs	Définition
JG	A > B	ZF = 0 et OF = SF	Jump if Greater
JNLE			Jump if Not Lower or Equal
JGE	A >= B	SF = OF	Jump if Greater or Equal
JNL			Jump if Not Lower
JNG	A <= B	(ZF = 1) ou (SF <> OF)	Jump if Not Greater
JLE			Jump if Lower or Equal
JNGE	A < B	SF <> OF	Jump if Not Greater or Equal
JL			Jump if Lower
JE	A = B	ZF = 1	Jump if Equal
JZ			Jump if Zero
JNE	A <> B	ZF = 0	Jump if Not Equal
JNZ			Jump if Not Zero

Exemple :

```
        MOV AH, 0FFh
        CMP AH, 0               ;compare FFh avec 0
        JL suite                ;test signé (saut si AH < 0)
        …
        …
    Suite :
        …
        …
```

L'instruction JL **provoquera le branchement** au label SUITE car –1 est inférieur à 0 car comme le test est signé, FFh est égal à –1.

4.9.3 Les boucles

4.9.3.1 LOOP

Cette instruction effectue une répétition tant que CX n'est pas égal à zéro; celle ci décrémente CX de 1 à chaque saut. Elle est l'équivalent de :

```
        Label1: DEC CX
                CMP CX, 0
                JNE Label1
```

Flags modifiés : Aucun

Exemple :

```
        MOV CX, 10
    Boucle_principale:
        …
        …
        Loop Boucle_principale
```

4.9.3.2 LOOPE (LOOP while Equal)

Cette instruction à la même fonction que l'instruction LOOP, elle décrémente de 1 CX, et compare CX avec 0, si ce n'est pas égale, elle saute au label spécifié, mais, de plus, elle saute au label que si l'indicateur ZF est égal a 1, donc si la dernière opération effectué a positionné cet indicateur.

Flags modifiés : Aucun

Exemple :

```
        ; Recherche d'un premier élément non nul
                MOV CX, LENGTH
                MOV SI, -1
        Suiv :  INC SI
                CMP TABLE[SI], 0
                LOOPE Suiv
                CMP CX, 0
                JNE Trouve
        PasTrouve :
                …
                …
        Trouve :
                …
                …
```

4.9.3.3 LOOPNE

Cette instruction décrémente CX et effectue un saut à l'adresse si CX <>0 et ZF = 0.

Flags modifiés : Aucun

Exemple : Voir LOOPE.

4.9.4 Les appels ou retours de sous-programmes

4.9.4.1 PROC (PROCedure)

Voir ENDP.

4.9.4.2 ENDP (END Procedure)

L'instruction PROC signale a l'assembleur le début d'un sous-programme et ENDP signale la fin des instructions du sous programme.
Les formes disponibles de ces instructions sont :

```
nomdeproc PROC NEAR
nomdeproc PROC FAR
nomdeproc ENDP
```

où « nomdeproc » est un identificateur représentant le nom que le programmeur donne au sous-programme.

Flags modifiés : Aucun
Exemple :

```
LireCar PROC NEAR    ;marque le début du sous-programme

        MOV AH, 0    ; AH ← service 0
        INT 16h      ; AL ← caractère lu
        RET

LireCar ENDP   ;marque la fin du sous-programme
```

4.9.4.3 CALL

En assembleur, on peu faire des sous-programmes (on est obligé, si on veux de la clarté, et si on veux alléger son programme). Cette instruction sert donc à "appeler", comme son nom l'indique, un sous-programme. Un peu plus technique, cette instruction va sauvegarder l'IP (Instruction Pointer), et va remplacer celui-ci par l'adresse du sous –programme (Proc).Le programme reprends sont cours à la fin du sous-programme.

Flags modifiés: Aucun

Exemple:

```
WriteMsg:
        PUSH AX
        MOV AX, 09h
        INT 21h
        POP AX
        RET

start:
        MOV DX, Offset Message
        CALL WriteMsg
```

Remarque : Les sous-programme modifient souvent les registres, alors prenez l'habitude d'utiliser PUSH et POP .

4.9.4.4 RET (RETurn)

Je vous parle de sous-programme, et bien cette instruction sert à terminer un sous programme, puis le programme reprends son cours normal, d'un point de vu technique, l'adresse de l'instruction est récupérée et placée dans IP.

Flags modifiés : Aucun
Exemple : Voir Call

4.9.4.5 IRET (Interrupt RETurn)

Marque la fin d'une interruption en récupérant IP, CS et les Flags puis continu l'exécution du programme à l'adresse récupérée.

Flags modifiés : AF- CF-DF-OF-SF-TF-ZF

4.10 Les opérations sur les chaînes

4.10.1 CLD (CLear Direction flag)

Cette instruction permet de positionner l'indicateur DF à 0

Flags modifiés : DF

4.10.2 STD (STore Direction flag)

Cette instruction permet de positionner l'indicateur DF à 1

Flags modifiés : DF

4.10.3 MOVSB (MOVe String Byte)

Déplace un octet de l'adresse DS:SI à l'adresse ES:DI, SI et DI sont ensuite incrémenté de 1 (si DF=0) ou décrémenté de 1 (si DF=1).

Flags modifiés : Aucun
Exemple :

```
Msg1    DB 'Salut'
Msg2    DB ?

MOV SI, OFFSET Msg1
MOV DI, OFFSET Msg2
MOVSB
```

Msg2 est maintenant égal à 'S'

Note :Voir instruction REP pour "améliorer" cette instruction.

4.10.4 MOVSW (MOVe String Word)

Déplace un mot de l'adresse DS:SI à l'adresse ES:DI, SI et DI sont ensuite incrémenté de 2 (si DF=0) ou décrémenté de 4 (si DF=1).

Flags modifiés : Aucun
Exemple : Voir MOVSB

4.10.5 MOVSD (MOVe String Double)

Déplace un mot de l'adresse DS:SI à l'adresse ES:DI, SI et DI sont ensuite incrémenté de 4 (si DF=0) ou décrémenté de 4 (si DF=1).

Flags modifiés : Aucun
Exemple : Voir MOVSB

4.10.6 LODSB (LOaD String Byte)

Cette instruction transfère le contenu de DS:SI dans AL, puis SI est incrémenté de 1 si DF=0, ou décrémenté de 1 si DF=1.

Flags modifiés : Aucun
Exemple :

```
Valeur  DB 69h
Valeur2 DB 70h

MOV AX, 0000h
CLD
MOV SI, OFFSET Valeur
LODSB
        (AX=0069h)
        (SI pointe maintenant sur Valeur2(SI+1)
LODSB
        (AX=0070h)
```

Voir instruction REP pour "améliorer" cette instruction.

4.10.7 LODSW (LOaD String Word)

Cette instruction transfère le contenu de DS:SI dans AX, puis SI est incrémenté de 2 si DF=0, ou décrémenté de 2 si DF=1.

Flags modifiés : Aucun
Exemple : Voir LODSB.

4.10.8 LODSD (LOaD String Double)

Cette instruction transfère le contenu de DS:SI dans EAX, puis SI est incrémenté de 4 si DF=0, ou décrémenté de 4 si DF=1.

Flags modifiés : Aucun
Exemple : Voir LODSB.

4.10.9 CMPSB (CoMPare String Byte)

Voir CMPSD.

4.10.10 CMPSW (CoMPare String Word)

Voir CMPSD.

4.10.11 CMPSD (CoMPare String Double)

Ces instructions comparent le contenu de l'adresse DS:SI ,et le contenu qui se trouve à ES:DI , on peut comparer un octet, un Mot ou un Double Mot. De plus, cette instruction incrémente ou décrémente SI et DI, suivant la valeur de DF (DF=0, SI et DI incrémenté; DF=1 SI et DI décrémenté).Si on utilise CMPSB, incrément/décrément de 1, CMPSW, de 2 ,et CMPSD de 4.

Flags modifiés : AF - CF - OF - PF - SF - ZF
Exemple :

```
SI=1234h
DI=4567h

CMPSW

SI=1236h
DI=4569h
```

4.10.12 STOSB (STOre String Byte)

Copie les données placés dans AL vers l'octet pointé par ES:DI. Le Registre DI est ensuite incrémenté de 1 si DF=0, et décrémenté de 1 si DF=1.

Flags modifiés : Aucun
Exemple :

```
Msg1    DB ?

MOV DI, OFFSET Msg1
MOV AL, 0
STOSB
```

Msg1 contient maintenant le message '0'

4.10.13 STOSW (STOre String Word)

Copie les données placés dans AX vers le mot pointé par ES:DI. Le Registre DI est ensuite incrémenté de 2 si DF=0, et décrémenté de 2 si DF=1.

Flags modifiés : Aucun
Exemple : Voir STOSB.

4.10.14 STOSD (STOre String Double)

Copie les données placés dans EAX vers le double pointé par ES:DI. Le Registre DI est ensuite incrémenté de 4 si DF=0, et décrémenté de 4 si DF=1.

Flags modifiés : Aucun
Exemple : Voir STOSB.

4.10.15 REP (REPeat)

Cette instruction est le gros avantage des instructions ci-dessus, l'opérande de cette instructions est une instruction. Le processeur effectue cette instructions, et décrémente le registre CX, jusqu'à ce que celui-ci soit nul. Il faut donc placé préalablement le "nombre de fois à répéter l'instruction dans le registre CX".

Flags modifiés: Aucun
Exemple :

```
Msg1    DB 'Salut'
Msg2    DB ?

MOV SI, OFFSET Msg1
MOV DI, OFFSET Msg2
MOV CX, 5
REP MOVSB
```

La chaîne Msg1 est maintenant identique à la chaîne Msg2.

Remarque : Cette instruction est utilisable qu'avec les instruction de manipulation de caractères.

4.10.16 REPE (REPeat while Equal)

Dérivé de l'instruction REP, cette instruction continu la répétition tant que CX n'est pas nul, ou tant que le contenu de ES:[DI] et le contenu de DS:[SI] sont identiques. Cette instruction est très intéressante pour comparer si deux chaînes sont identiques.

Flags modifiés : Aucun
Exemple :

```
Msg1    DB 'Salut'
Msg2    DB 'Salue'

MOV SI, OFFSET Msg1
MOV DI, OFFSET Msg2
MOV CX, 5
REPE MOVSB
```

La répétition s'arrêtera au dernier caractère ("t" et "e" sont différent). On peut donc voir le nombre de caractères identiques en regardant l'état de CX, qui est décrémenté comme avec l'instruction REP.

4.10.17 REPNE (REPeat while Not Equal)

Toujours un dérivé de REP, mais cette fois, contrairement à REPE, au lieu que la répétition s'arrête quand deux caractères sont différents, la répétition s'arrête quand deux caractères sont identiques.

Flags modifiés : Aucun
Exemple : Voir REPE.

4.10.18 REPZ (REPeat while Zero)

Identique à REPE.

4.10.19 REPNZ (REPeat while Not Zero)

Identique à REPNE.

4.10.20 SCASB (SCAn String Byte)

Compare l'octet pointé par ES:DI, avec le registre 8 bits AL. Le Registre DI est ensuite incrémenté de 1.

Flags modifiés : AF - CF - OF - PF- SF - ZF
Exemple :

```
Msg1    DB 'Salut'

MOV DI, OFFSET Msg1
MOV AL, 'l'
REPNE SCASB
```

La répétition s'arrêtera au 3ème caractères, le caractère l.

4.10.21 SCASW (SCAn String Word)

Compare le mot pointé par ES:DI, avec le registre 16 bits AX. Le Registre DI est ensuite incrémenté de 2.

Flags modifiés : AF - CF - OF - PF- SF - ZF
Exemple : Voir SCASB.

4.10.22 SCASD (SCAn String Double)

Compare le double pointé par ES:DI, avec le registre 32 bits EAX. Le Registre DI est ensuite incrémenté de 4.

Flags modifiés : AF - CF - OF - PF- SF - ZF
Exemple : Voir SCASB.

4.11 Les instructions de gestion des adresses

4.11.1 LEA (Load Effective Address)

Cette instruction transfère l'adresse d'Offset de la source dans le registre de destination. (c'est donc l'équivalent de MOV REG, OFFSET source), pour des registres de type index ou base.

Flags modifiés : Aucun

Exemple :

LEA SI, Table ⇔ MOV SI, OFFSET TABLE

4.11.2 LES (Load Extra Segment)

Charge une adresse 32 bits dans le couple de registres constitué de ES:Registre :

ES → contient le segment
Registre → contient l'offset

Flags modifiés : Aucun
Exemple :

LES SI, Pointeur

4.11.3 LDS (Load Data Segment)

Charge une adresse 32 bits dans le couple de registres constitué de DS:Registre :

DS → contient le segment
Registre → contient l'offset

Flags modifiés : Aucun
Exemple :

LDS DI, Pointeur

4.12 Déclaration de données

4.12.1 DB (Define Byte)

Cette instruction sert à définir un octet, ainsi vous pouvez définir un nombre inférieur à 255, une chaîne de caractères, ...

Flags modifiés : Aucun
Exemple :

```
Mess1       DB 'Salut à toi','$'
Nbr1        DB 69
ColorScr    DB ?
```

La premier label contient la chaîne de caractère « salut à toi$ », le second le nombre décimal 69, et le dernier contient un nombre indéfini, on s'en servira dans le programme pour remplacer ce nombre par un nombre connu.

4.12.2 DW (Define Word)

Cette instruction définit un mot, donc une valeur inférieur ou égale à 65535.

Flags modifiés : Aucun
Exemple : Voir DB.

4.12.3 DD (Define Double)

Cette instruction définit un double mot (32 bits).

Flags modifiés : Aucun
Exemple : Voir DB.

4.12.4 DQ (Define Quadword)

Cette instruction définit un quadruple mot (64 bits).

Flags modifiés : Aucun
Exemple : Voir DB.

4.12.5 EQU (EQUivalent)

Cette instruction affecte un nombre à un label. La valeur ne peut pas être modifiée : c'est une déclaration de constante.

Flags modifiés : Aucun
Exemple :

KbEsc EQU 283

Chapitre 5

5 Exemples

5.1 Pour débuter

```
Data SEGMENT
      T     DB 100, 02Ah, 31
      V1    DB 5, 6
      V2    DW 0A654h
      M     DB 'CHAINE DE CARACTERES'
      V3    DB 4
Data ENDS
Code SEGMENT
      Assume CS : Code, DS : Data
    Deb :
      MOV AX, Data
      MOV DS, AX
      MOV BL, V1
      ADD V3, BL
      MOV BH, 4
      INC BH
      MOV T, BH
      MOV AH, 4Ch
      INT 21h
Code ENDS
      END Deb
```

5.2 Transfert de chaînes

```
Donnees SEGMENT
      M1            DB 'origine'
      M2            DB 7 DUP¨('*')
      Longueur      DB 0
Donnees ENDS

Code SEGMENT
      Assume CS : Code, DS : Donnees
    Entree :
      MOV AX, Donnees
      MOV DS, AX
      MOV Longueur, OFFSET M2
      MOV CX, 0
      MOV CL, Longueur
      LEA SI, M1
      MOV DI, OFFSET M2
    Trans :
      MOV AL, [SI]
      MOV [DI], AL
      INC SI
      INC DI
      LOOP Trans
      MOV AH, 4Ch
      INT 21h
Code ENDS
      END Entree
```

5.3 Ecriture directe à l'écran

```
Code SEGMENT
      Assume CS : Code, DS : Donnees
    Deb :
      PUSH CS
      POP DS
      JMP La_bas
    Ecran_Mono      DW 0B000h
    Etoile          DB '*'
    Las_bas :
      MOV AX, Ecran_Mono
      MOV ES, AX
      MOV SI, 0
      MOV CX, 2000
      MOV AL, Etoile
    Encore :
      MOV ES :[SI], AL
      INC SI
      INC SI
      LOOP Encore
      MOV AH, 4Ch
      INT 21h
Code ENDS
      END Deb
```

5.4 Lecture de chaîne et affichage

```
Data SEGMENT
       Mess1       DB ‘Entrez une chaîne’
                   DB 10, 13, ‘$’
       Buffer      DB 40, 0, 40 DUP (‘$’),’$’
Data ENDS
Code SEGMENT PUBLIC
      Assume CS : Code, DS : Data
    Debut :
      MOV AX, Data
      MOV DS, AX
      MOV DX, OFFSET Mess1
      MOV AH, 9
      INT 21h
      LEA DX, Buffer
      MOV AH, 10
      INT 21h
      MOV DX, OFFSET Buffer + 2
      MOV AH, 9
      INT 21h
      MOV AH, 4Ch
      INT 21h
Code ENDS
      END Debut
```